EXTENDING THE METANET NETWORK ENVIRONMENT: PROCESS CONTROL AND MACHINE INDEPENDENCE

Jacob M.J. Murre and Steven E. Kleynenberg

Leiden University, Unit of Experimental and Theoretical Psychology
P.O.Box 9555, 2300 RB Leiden, The Netherlands, E-mail: MURRE@HLERUL55.Bitnet

Abstract This study reports on a project to build a general, parallel neurosimulator (GPN), i.e., a simulator that is consistently extensible with novel network paradigms and parallel hardware. Our system is called the *MetaNet Network Environment* [16]. Four problems in the construction of a GPN are discussed. They concern the representation of objects and object interdependencies, concurrent process control, and machine independence. The current state of the MetaNet system is described, followed by a discussion of an extended version based on four levels of description. We introduce a low-level process description that is to be viewed as instructions for an ideal, virtual, parallel machine.

1. INTRODUCTION

In 1988, the predecessor of the *MetaNet Network Environment* was developed [16], because no existing neurosimulator could adequately handle the modular type of neural networks developed by our group (i.e., Categorizing And Learning Module, see [18], [19]). The development of many neurosimulators seems to have been motivated by similar reasons. An informal search through the literature revealed no less than 45 neurosimulators (listed in [17]), but only a small number of these supports many different paradigms and many types of parallel hardware. Few neurosimulators (mostly commercial systems) allow the user to extend the system in a consistent manner, for example, by adding a new neural network paradigm or parallel hardware to the available range. In the few consistently extensible neurosimulators, added paradigms often lack the (graphical) interaction possibilities of the 'standard' paradigms, such as revealing the internals of a model (e.g., see review of three important commercial systems by [2]).

The MetaNet Network Environment [16] was conceived as a *general, parallel neurosimulator*. Such a system is faced with a demanding set of problems, because the neural network family includes paradigms with a wide variety of characteristics, and, in addition to this, parallel hardware exhibits a similar range of idiosyncrasies. Mapping a certain neural network onto some type of parallel hardware can, therefore, take many different forms. The situation is aggravated by the fact that users of neurosimulators may range from relative novices to expert users. In particular, we believe that the following four problems are major obstacles for the development of a general, parallel neurosimulator.

(1) *The representation problem*. How can a problem (i.e., a neural network) be decomposed into a representation that is both manageable and consistent? In our original approach to this problem, MetaNet was structured using a fixed object hierarchy (i.e., MODEL -> MODULE -> GROUP, etc., see next section). Most recent neurosimulators have adopted a comparable, object-oriented strategy.

(2) *The dependency problem*. How can relations between constituent parts of a problem representation be made explicit? The interdependencies between objects in most existing simulators are not handled in a consistent manner. Most systems allow for specific connections to be constructed (e.g., between any two nodes and between layers). Some offer a range of connection schemes that can be applied to only a single kind of object. These connections are of the type typically found in neural networks (i.e., directed connections that have an adaptable, real weight as attribute). In MetaNet, any two objects of the same type can be connected by generalized connections called TRACTs (explained below).

(3) *The control problem*. How can a problem representation be converted to a process description without losing potential parallelism? Some readers might wonder whether the control problem applies at all to neural networks. After all, they are neurally inspired structures and should not be subject to some type of central control. In reality, however, many networks are not as 'neural' as is to be desired. For example, the popular backpropagation model [22] has a definitely 'non-neural' control structure. In a model with a fully 'neural' control structure, all nodes process their activations in parallel, without any 'sweeps' or other explicit synchronization.

We stumbled on the control problem, while trying to devise an environment in which users can define networks with novel topologies, parameters, and activation and learning dynamics. The problem occurs when the user also wants to constrain the order in which the distinguished

objects are to be processed (cf. layers in a backpropagation network). A possible solution to this problem is simply to prompt the user to specify such an order. If no special precautions are taken, however, such a linear ordering may result in partial or total loss of parallelism. Another option is to have the user write out the control structure in some kind of programming language that supports parallelism, for example, using some form of concurrent C. For some purposes this approach may be adequate, but because the principal idea of a neurosimulator - or any simulator for that matter - is to free the user from as many distracting details as possible, this can never be a general solution. Specifying an explicitly parallel control structure will be an extra burden for most users, that is alien to the actual task: solving a certain problem using neural networks. It should, therefore, be avoided. Our proposed solution to this problem is based on the introduction of several levels of description. A network is converted from a problem oriented description into a high-level concurrent process description. This code, in turn, is translated into an intermediate code, that is a low-level description of concurrent processes. (See Section 3.2 for a more detailed exposition).

(4) *The machine problem*. How can a process description (i.e., of a certain simulation) be optimally mapped onto a parallel machine? Most parallel neurosimulators deal with machine dependencies in an ad hoc manner. This means implementing each specific type of network (e.g., backpropagation) on each available specific type of parallel hardware (e.g., a transputer network), so that the user is still forced to explicitly map a novel type of network onto the supported parallel hardware. Indeed, at present it does not seem feasible to completely shield the user from the machine level, especially if a wide range of parallel machines is to be used. If the concurrency in the process description of a certain model is specified in a sufficiently general format, however, it seems possible to convert (automatically) such intermediate code into a low-level machine dependent code for many different machines (e.g., into Occam for a transputer network). In Section 3.2, we will propose using a concurrent process description of neural networks as intermediate code. The intermediate code is viewed as an ideal, virtual machine, that is emulated by the host hardware. Interpretation of the intermediate code is done by hardware drivers.

Before discussing a novel approach to problems (3) and (4) by the extended MetaNet Network Environment, we will first briefly describe the current state of the system.

2. The METANET Network Environment

The MetaNet user interface is based on two familiar design concepts, it is menu-driven and graphics oriented. Interaction with the system is carried out by use of a mouse. The MetaNet screen is divided into three sections (see Fig. 1): a menu section on the right side, a message section at the bottom, and a large central section for network editing. The edit section is a viewport on an essentially unbounded two-dimensional underlaying workspace, in which networks can be constructed. The interface supports scrolling and zooming facilities, allowing the user to view the workspace in any detail, from any viewpoint.

In the current implementation, MetaNet imposes a fixed object hierarchy on a network topology. Four levels are distinguished. The lowest level is formed by NODES. These combine into GROUPS, which assemble into MODULES. These in turn form MODELS.

Constructing a network in MetaNet is done in a top-down fashion, starting at the highest MODEL level and graduately working down to the NODE level. The user first marks a rectangular area of arbitrary size in the workspace. This area becomes a new MODEL. MetaNet allows for the construction and running of multiple models simultaneously, so the user can start by setting up more than one MODEL. Having defined one or more (empty) MODELs in the workspace, MODULES can be placed inside the MODELs. The MODULE forms an important anchoring point in MetaNet. At this level the user can select from standard network architectures (e.g. backpropagation, CALM, Hopfield, etc.) along with learning and control paradigms. If a standard paradigm is chosen, the MODULE will be automatically filled with a standard configuration of GROUPs. In all cases, however, the user may change this configuration (e.g., by deleting GROUPs) or define completely new MODULEs.

A GROUP consists of NODEs which have to be of the same type, and can be arranged in an $n \times m$ matrix (i.e., a GROUP may be a column, row, or full matrix). By superimposing a GRID upon the screen, where each of its points indicate a NODE postition, precise control over the size and the location of the GROUPs is obtained. Once a number of GROUPs is in place, they can be connected by using the mouse to indicate source and target groups. The set of all NODE-to-NODE CONNECTIONs between two GROUPs is called a TRACT. TRACTs are generalized connections, and actions taken upon TRACTs affect all its constituent CONNECTIONs in an uniform way. By default, the system displays TRACTs, suppressing the multitude of underlaying CONNECTIONs. The individual CONNECTIONs can easily be made visible and may be modified, deleted, etc. TRACTs may be formed with different wiring-schemes, to be selected from a menu, such as as One-To-One, All-To-All, and more complex forms. Because it is possible to connect a NODE in a GROUP to other nodes in the same GROUP (e.g., connecting each node to its adjacent ones), several wiring-schemes are incorporated that particularly facilitate the creation of cellular automata, such as the Von Neumann and Moore neigbourhoods. In a similar fashion, lateral inhibition can be specified.

The principal objects in MetaNet form two different classes: the Unit and the Connector class. The Unit class consists of MODELs, MODULEs, GROUPs, and NODEs. Actions that can be applied to all the members of a class are called its General actions, for the

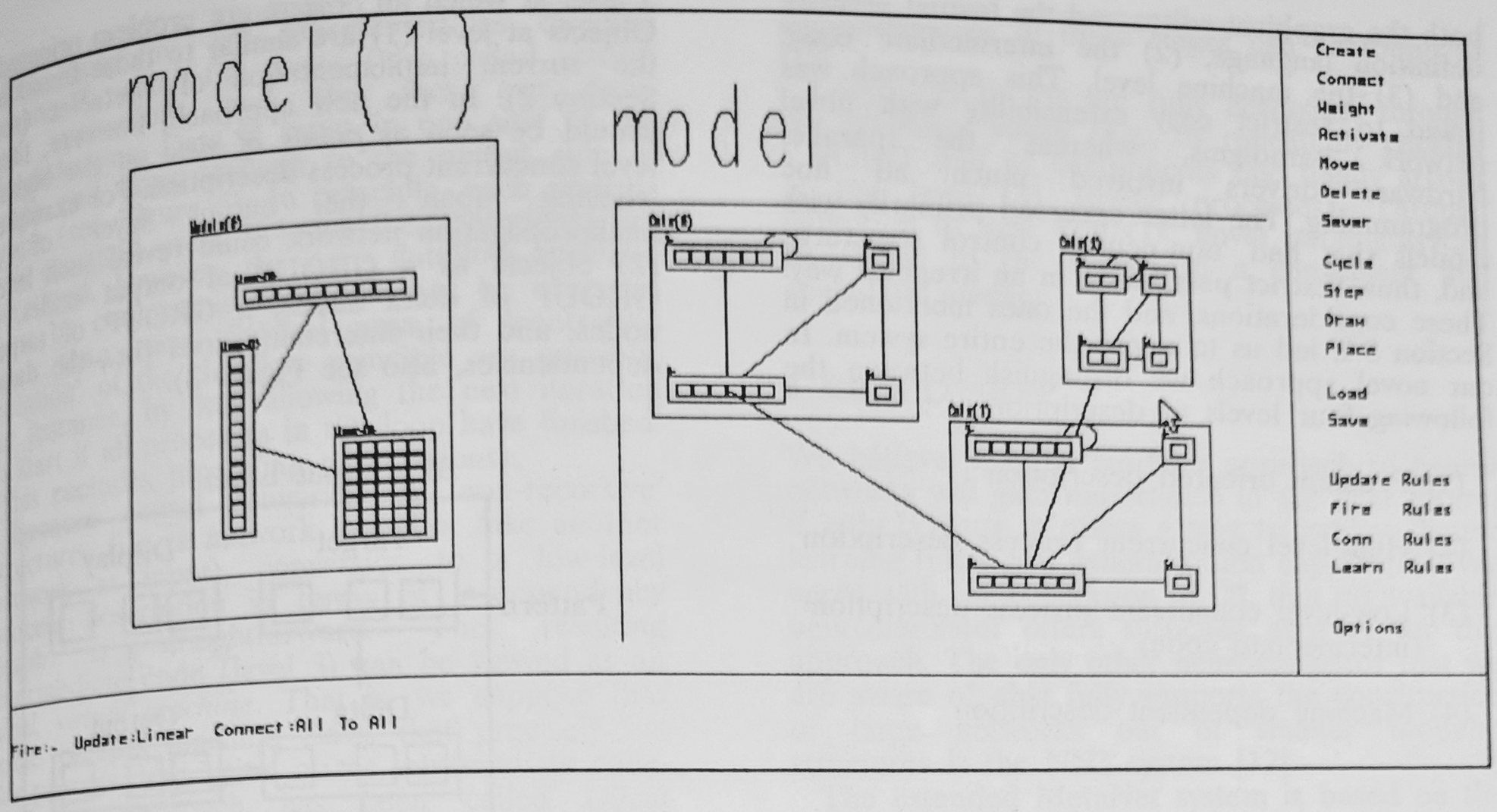


Fig.1. The MetaNet screen.

Unit class these actions include: create, delete, modify, move, copy, clamp, reset, load, and save. The Connector-class, consisting of TRACTs and CONNECTIONs, can be manipulated through the following General actions: connect, sever, modify, copy, clamp, and reset. In contrast to General actions, a Specific action is only applicable to an individual member of a class. An example of a Specific action would be activating a NODE. Together these functions enable the user to define any two-dimensional network topology, limited only by the amount of memory available to the system.

Due to its modular structure, apart from saving an entire network, parts of it can be saved separately. In this way, libraries of user-defined network components can be constructed, which can be loaded into the system at any time.

All objects carrying parameters, like the weight of a CONNECTION or the activation of a NODE can be inspected and modified. By clicking an object with the mouse, all relevant parameters are displayed in a popup window, and are directly modifiable. MetaNet uses color-scales (or gray-scales, depending on the video display unit) to indicate NODE activations or CONNECTION weights. In this way, the dynamics of a running network become clearly visible. Other display options are weight matrices, topological maps, and state graphs. The system, furthermore, has options such as the possibility to induce noise on connections or to randomly activate GROUPs of NODEs.

As stated above, networks or parts of it can be saved for later use. MetaNet stores a network's blueprint in a Network Description File (NDF). The NDF has an ASCII format and may be written or read by other programs. Also, the NDF can be translated into source code for some standard programming languages. At the moment, only translation into C is supported, but future releases will also include C++ and LISP.

3. DISCUSSION

3.1 Evaluation of MetaNet

The UNIT hierarchy in MetaNet functions well, and so does the option to connect UNITs with TRACTs. Starting from scratch, it is possible to rapidly create networks with complicated topological structures. This is particularly true for modular neural networks. The ability to zoom in on any part of the (essentially infinite) work space and drag around (and delete, 'click open', etc.) anything in view allows for a flexible interaction. The option to suppress detail (e.g., by showing TRACTs rather than the individual connections) further supports easy handling of networks. Because we have reduced the use of circles, text, and dotted lines in the graphical editor, and because we use a very fast graphics kernel, the current implementation of the graphical editor (written in C) gives a fast response, even on modest PCs.

Extending the system with totally novel types of networks, however, presents problems. This is especially true for models that constrain the order in which objects are to be processed, such as layers in backpropagation. Adding such a control structure, currently, involves some ad hoc programming. Specifying networks using the textual network definition language in MetaNet [16] does not smoothly blend with the operation of the graphical interface. The graphical interface, however, does not allow specification of low-level details, such as non-standard convergence checks or activation rules. As will be described below, because of these drawbacks, we are no longer further developing a textual network definition language. Instead, we now aim at a consistently graphical system, with a high-level *graphical* programming language.

3.2 Toward a general, parallel neurosimulator

In our original approach we distinguished between three levels of description: (1) the network definition level, which coincided with

both the graphical editor and the textual network definition language, (2) the intermediate code, and (3) the machine level. This approach was found to restrict easy extensibility with novel network paradigms, whereas the parallel hardware drivers involved much ad hoc programming. The latter occurred primarily with models that had 'non-neural' control structures and, thus, restrict parallelism in an irregular way. These considerations, and the ones mentioned in Section 3.1, led us to revise the entire system. In our novel approach we distinguish between the following four levels of description.

(1) Problem oriented description

(2) High-level concurrent process description

(3) Low-level concurrent process description (intermediate code)

(4) Machine dependent description

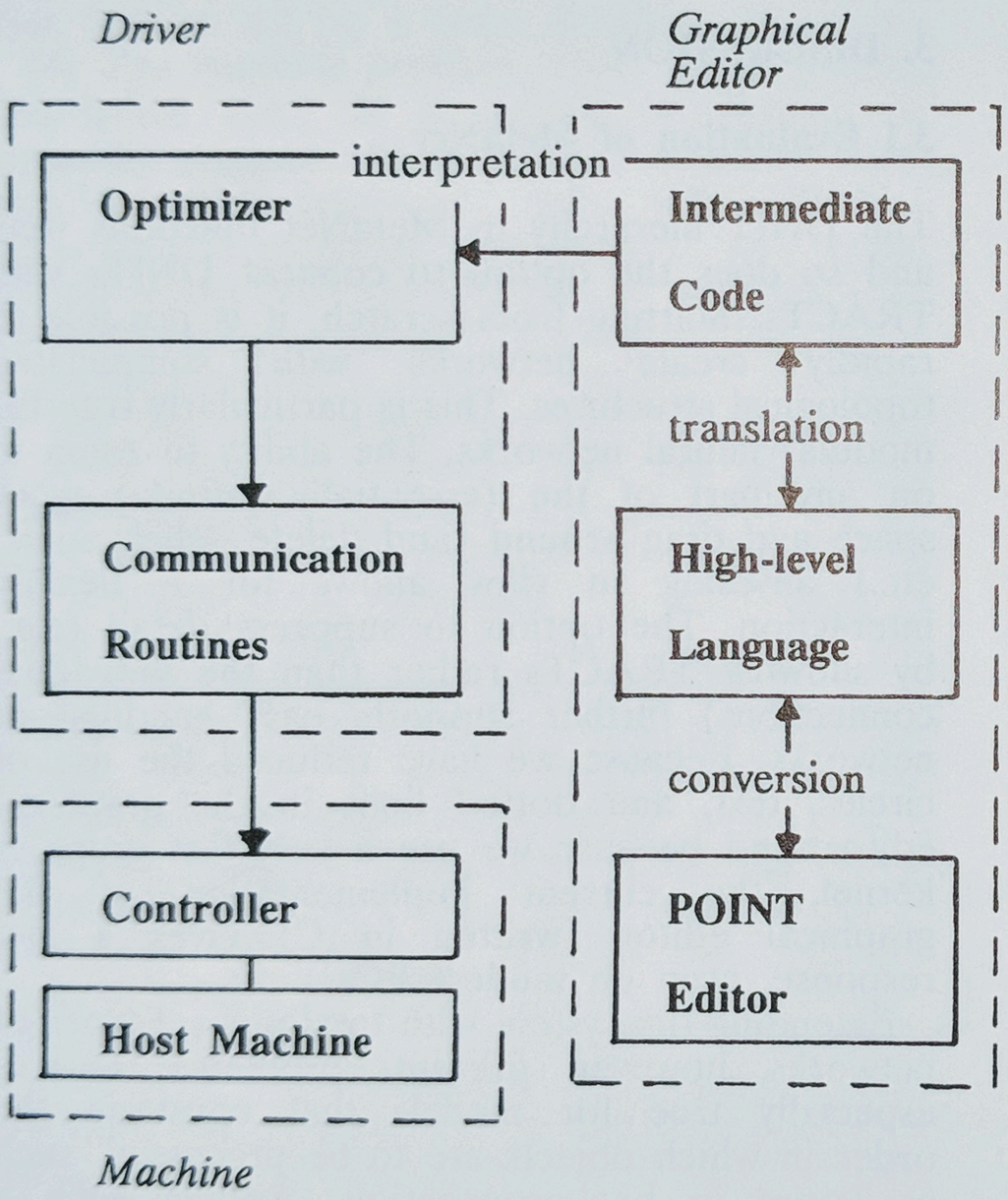


Fig.2. Overview of the extended system.

Levels (1) to (3) are handled by a graphical editor (also see Figure 2), so that all levels of description can be approached in single style of interaction, closely approximating direct-manipulation (e.g., [24], [25]). The intermediate code allows for a rigorous specification of a problem (i.e., a simulation) in terms of concurrent low-level or atomic processes. Specifying detailed low-level code is usually done indirectly, through some higher level language. Here, we have opted for a graphical higher level language (see below). Because even a higher level concurrent process description contains too many details that are irrelevant for the problem at hand (i.e., getting a neural network to run on some possibly parallel machine), we have added a level at which all objects are problem oriented. Objects at level (1) are similar to those found in the current implementation of MetaNet (see Section 2). In the new approach, however, they should be seen as *points of view* on the higher level concurrent process description. For example, 'clicking open' the output layer of a backpropagation network could reveal such level (2) objects as a GROUP of output nodes, a GROUP of delta nodes, a GROUP of target nodes, and their interconnections (i.e., the data-dependencies, also see Figure 3).

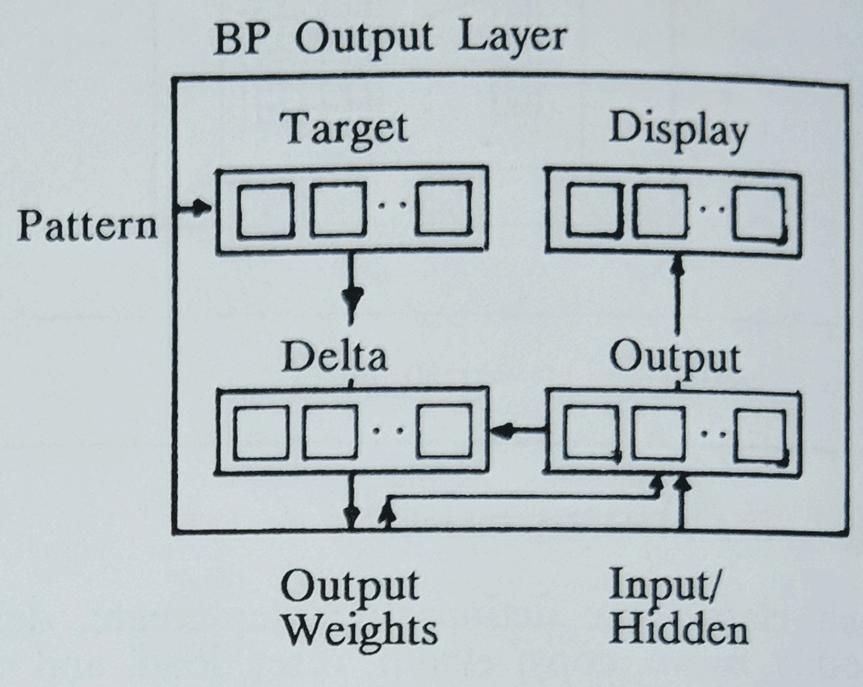


Fig.3. A POINT, clicked open to reveal level (2) internal objects.

At level (1) we provide for a set of Problem Oriented INteraction Tools (abbreviated to POINTs, i.e., points of view on the lower levels). New POINTs may be defined in terms of level (2) objects, so that the system can be consistently extended with novel paradigms.

Levels (2) and (3) are based on a concurrent process description. There exists an extensive literature on concurrent processes, and several formal methods have been proposed for representing concurrency problems, for example, Petri-nets [20], [21], dependency graphs, and trace theory [13], [1]. It has been proved that all of these theories can be expressed within the more general framework of process algebra [14]. An example of a dependence graph is shown in Figure 4. If two processes are connected by an arrow, for example, *A* -> *B*, this means that process *B* cannot start until

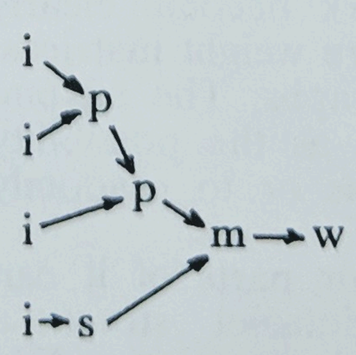


Fig.4.

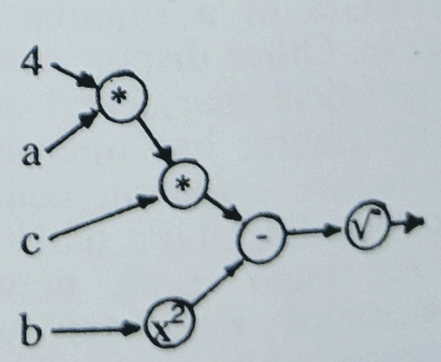


Fig.5.

process *A* has finished. Based on this simple principle, a representation of any problem can be made that fully captures the potential parallelism.

A computing principle that rather closely follows this approach to representing concurrency is dataflow (e.g., [3], [8], and [7] for reviews).

Here, programs are represented as directed graphs, in which the nodes (also called actors) are instructions and the arcs are data-dependencies. Data is input in one side of the graph and flows through to the output side of graph (see Figure 5). In principle, each node is only executed once. If loops (i.e., iterations) are permitted, however, the pure dataflow principle must be compromised; nodes may be executed more than once. Several ways of 'controlled violation' of the dataflow principle are possible, for instance, by only allowing the next iteration to start if all processes in the loop have finished. (This excludes pipelining of iterations).

Because neural networks are 'non-recursive' structures (i.e., a network does not take another network as input), conversion to a low-level process description in terms of a dependency graph is straightforward. The resulting intermediate code (level 3) can be viewed as an *ideal, virtual machine*. That is, we suppose that some parallel machine exists that precisely fits the process description of the intermediate code. Such an approach has been called *Direct Correspondence Architecture* ([4], [5], [10], [28]). The intermediate, level (3) code in MetaNet can be viewed as a *Directly Executed Language* (DEL). Translation from the level (2) description to the DEL at level (3) is simple, so that the intermediate code can be characterized as oriented towards the high level description, rather than to the particularities of some destination machine.

At the machine level (4), the ideal, virtual machine must be emulated by some available hardware. This means that the level (3) code must be interpreted for some host machine (e.g., a transputer network). Because the host machine may be a parallel system with a complicated interprocessor communication, the interpretation can take many forms. There may also be an optimization stage in which processor loads and communication overhead are adjusted, so that total iteration time is minimized. For example, on a transputer network with known topology, the interpretation roughly corresponds to an assignment of level (3) processes to transputers. For modular neural networks (but not for fully connected networks, see [15]), it may be advantageous to position the processes in such a manner, that communication overhead is minimized (e.g., every node could represent a process). This problem is in fact a graph K-partitioning problem. For the case where we had only two transputers, the problem would be to divide the neural network into two equal parts such, that the number of interconnections between the parts are minimized. This class of problems is well known to be NP-complete, so that some sub-optimal algorithm must be used.

Let us summarize the stages through which a network model passes in the extended MetaNet system (also see Figure 2). (1) A network model is assembled by the user from a number of POINTs. (2) The model is converted into a high-level description. (3) The description is translated into low-level, intermediate code. (4) This code is interpreted and optimized for a certain host machine (by a high-level driver). Communication with the host is maintained through a set of communication routines (a low-level driver). Process scheduling and synchronization is handled by the machine controller. Further details on the implementation of these stages can be found in [17].

Levels (1) to (3) are fully accessible through the graphical editor (e.g., by repeatedly clicking a POINT and its components one finally arrives at the level (3) atomic processes). Through definition of new POINTs, novel paradigms can be added to the system in a convenient and consistent manner.

4. CONCLUSION

We believe that the modular approach to neural networks will gain importance in the near future, if only because it offers a way to achieve better learning times and generalization capabilities. We agree with the conclusion of [2], that no available neurosimulator offers sufficient support for this approach. The only other neurosimulator that we are aware of, that fully supports the construction of large networks out of smaller modular structures is the NN3 system [12].

The extended MetaNet system is based on the idea of an ideal, virtual machine. A natural next step would be to develop a machine that approximates this ideal as closely as possible. This is indeed our intention. At the moment we are interfacing MetaNet with a dedicated 400 processor neurocomputer, that is based on the principle of modularity [9]. We have recently started a new project, aimed at developing an extended version of this machine, that comes closer to the 'ideal neurocomputer'.

As described above, an important task for the high-level drivers is to optimize the interpreted level (3) code for processing and communication loadbalancing. Part of this problem can be reduced to a graph K-partitioning problem. Recently, a number of authors have applied neural networks to the problems of graph partitioning [27] and direct loadbalancing [23]. A logical step would, thus, be to implement the high-level driver as a neural network, in this way achieving something of a 'neural bootstrap'. Similarly, the user interface itself could be implemented in terms of fine-grained parallel structures (see for instance [26]), some of which could well be neural networks used to assist the user in certain tasks (e.g., predicting the next chosen menu item). Finally, the conversion, translation, and interpretation could be implemented in fine-grained parallelism, for example, through parallel graph rewriting.

Though we might well wonder whether a General Neurocomputer (GNC) is a possibility, we feel that only a unified approach to user interaction and integration of computational resources will lead towards such a goal. Ultimately, this implies that neurocomputers must be designed to serve as ideal machines for neurosimulators.

References

[1] Aalbergsberg, IJ.J., & G. Rozenberg (1988). Theory of traces. *Theoretical Computer Science*, 60, 1-82.

[2] Cohen, H. (1989). How useful are current neural network software tools? *Neural Network Review*, 3, 102-113.

[3] Computer (1982). Special issue on dataflow.

IEEE Computer, 15(2).

[4] Flynn, M.J. (1980). Directions and issues in architecture and language: Language -> Architecture -> Machine. *IEEE Computer*, 13 (10), 5-22.

[5] Flynn, M.J., & L.W. Hoevel (1979). *A theory of interpretive architectures: Ideal language machines*. Technical Report 170, Stanford University.

[7] Gaudiot, J.L. (1989). Dataflow machines. In: V.M. Milutinovic (Ed.), *High-level language computer architecture*, Computer Science Press.

[8] Gurd, J.R. (1984). Fundamentals of dataflow. In: F.B. Chambers, D.A. Duce, and G.P. Jones (Eds.), *Distributed Computing*, London: Academic Press.

[9] Heemskerk, J.N.H, J.M.J., Murre, & P.T.W. Hudson (1991). The BSP400: a modular neurocomputer, assembled from 400 low-cost microcomputers. *Proceedings of the ICANN-91-Helsinki.*

[10] Hoch, J. & C. Bauer (1989). The DELtran Project. In: V.M. Milutinovic (Ed.), *High-Level Language Computer Architecture*. Computer Science Press.

[11] Kennaway, J.R., & M.R. Sleep (1984). The 'language first' approach. In: F.B. Chambers, D.A. Duce, & G.P. Jones (Eds.), *Distributed Computing*. London: Academic Press.

[12] Linden, A, & Ch. Tietz (1991, in prep.). *Research and development software environment for modular adaptive systems*. Technical Report NN3-1. GMD Birlinghoven, Sankt Augustin, Germany.

[13] Mazurkiewitz, A. (1977). *Concurrent program schemes and their interpretations*. DAIMI Rept. PB-78, Aarhus University, Aarhus.

[14] Mazurkiewitz, A. (1984). Traces, histories, graphs: instances of a process monoid. *Lecture Notes in Computer Science*, 176, 115-133.

[15] Murre, J.M.J. (1991). Transputer implementations of neural networks: an analysis. *Proceedings of the ICANN-91-Helsinki.*

[16] Murre, J.M.J., & S.E. Kleynenberg (1990). The MetaNet Network Environmnent for the development of modular neural networks. *Proceedings of the INNC-90-Paris*, 717-720.

[17] Murre, J.M.J., & S.E. Kleynenberg (in prep.). *Process control and simulation management in general, parallel neurosimulators.*

[18] Murre, J.M.J., R.H. Phaf, & G. Wolters (1989). CALM networks: a modular approach to supervised and unsupervised learning. *Proceedings of the IJCNN-89-Washington*, 1, 649-656.

[19] Murre, J.M.J., R.H. Phaf, & G. Wolters (submitted). *CALM: Categorizing And Learning Module*. Leiden Psychological Reports, ETP 02-90, Leiden University, Leiden.

[20] Petri, C.A. (1962). *Kommunikation mit Automaten*. Schriften des Institutes für Instrumentelle Mathematik, Bonn.

[21] Reisig. W. (1982). *Petrinetze: Eine Einführung*. Berlin: Springer-Verlag.

[22] Rumelhart, D.E., G.E. Hinton, & R.J. Williams (1986). Learning internal representations by error propagation. In: Rumelhart, D.E., & J.L. McClelland (Eds.), *Parallel distributed processing. Volume 1: Foundations*. Cambridge, MA: MIT Press.

[23] Schmid, P. (1990). The mapping problem: a neural network approach. *Proceedings of INNC-90-Paris*, 274-277.

[24] Shneiderman, B. (1983). Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16, 57-69.

[25] Shneiderman, B. (1987). *Designing the user interface: strategies for effective human-computer interaction*. Reading, MA: Addison-Wesley.

[26] Van den Bos, J. (1988). Abstract Interaction Tools: A language for user interface management systems. *ACM Transactions on Programming Languages and Systems*, 10, 215-247.

[27] Van den Bout, D.E., & T.K. Miller III (1990). Graph partitioning using annealed neural networks. *IEEE Transactions on Neural Networks*, 1, 192-203.

[28] Wedig, R.G. (1989). Direct correspondence architectures: principles, architectures, and design. In: V.M. Milutinovic (Ed.), *High-level language computer architecture*. Computer Science Press.