

# A Framework for the Merger and Practical Exploitation of Formal Logic and Artificial Neural Networks

Gareth Howells and Konstantinos Sirlantzis

**Abstract**—The assimilation of formal logic into the domain of Software Engineering offers the possibility of enormous benefits in terms of software reliability and verifiability. To date, however, the integration of such techniques has proved difficult since they involve a significantly increased burden on the programmer in meeting the demands of the formal mechanisms being employed. The current paper investigates the advantages which may be gained by the software development process with the introduction of Artificial Neural Network technology into a formal software development system. Essentially, the adaptive artificial neural network model is employed to refine an existing formal software model in order to produce increasingly better approximations to a given solution. Each approximation is itself a valid formal system whose precise behaviour may be formally determined. The paper introduces a framework by which a programmer may define a system possessing the abstract structure of a traditional neural network but whose internal structures are taken from the formal mathematical domain of Constructive Type Theory. The system will then refine itself to produce successive approximations to a desired goal based on data presented to it. An example is presented addressing a problem domain which has previously proved difficult to model. Although the example presented is necessarily limited, it does provide an insight into the potential advantages of merging formal logic with artificial neural systems..

**Index Terms**—Formal Logic, Constructive Type Theory, Neural Networks.

## I. INTRODUCTION

The application of Formal Logic in the form of Constructive Mathematics such as *Constructive Type Theory* [6,13] to the design of major software systems can give rise to significant improvements in the reliability and dependability of the software produced.

The major factors hindering the widespread application of such techniques lie in the difficulty encountered in deriving the logical statements (*Judgements*) defining the software system and inefficiencies in the implementation of the system so derived. In essence, improvements in the

reliability of the final system are offset by costs associated with the initial formal derivation of the system.

The purpose of this paper is to explore the relationship between Constructive Type Theory and Artificial Neural Networks by illustrating how a traditional example network may be implemented in a system derived from Type Theory. The development framework employed in the examples retains many of the advantages offered by Constructive Type Theory whilst removing some of the burden of design from the programmer and placing it on generic learning rules taken from the domain of Artificial Neural Networks. That is, the learning rules of the neural network will perform some of the work traditionally performed by the programmer. The design framework proposed offers the following advantages:

- The construction of formally derived and verifiable software system related to the domain of artificial neural networks without requiring detailed mathematical analysis by the programmer.
- The derived system may adapt itself to take into account differing future performance requirements within the limitations of the network architecture simulated.

The application area of the framework is typically envisaged to be one possessing a significant amount of sample data which may be used for the neural network refinement (learning) process. A problem domain where the data is multidimensional in nature is especially suitable as it provides the opportunity to demonstrate the applicability of the system in areas which prove difficult to analyse using existing techniques. A problem domain meeting the above conditions is the derivation of a mathematical model to allow predictions to be made regarding variables pertaining to Dover Harbour in Kent, U.K. The variables governing the behaviour of the harbour are numerous and the harbour itself displays a complex, chaotic behaviour which has, to date, defied the construction of an accurate model

## II. CONSTRUCTIVE TYPE THEORY

*Constructive Type Theory* is a formal logic [6,7,8,13] and it is based on *Constructive Mathematics* in which proofs must be based on a demonstration of how to construct an example of the theorem or proposition being asserted. In other words, proofs by contradiction are not allowed. So, in Constructive Type Theory *proofs* can be thought of as *algorithms* to create an example of the *proposition* in hand. Furthermore, the proposition forms a *datatype definition* or at a higher level a *formal specification* for the algorithm itself.

In Constructive Type Theory, each logical proposition is accompanied by its proof object forming a pair of values

Manuscript received February 25, 2009. This research was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant No. GR/L78338..

Gareth Howells is with the Department of Electronics at the University of Kent, Canterbury Kent, UK (phone: +44 1227 823724; fax: +44 1227 456084; e-mail: W.G.J.Howells@kent.ac.uk).

Konstantinos Sirlantzis is with the Department of Electronics at the University of Kent, Canterbury Kent, UK (e-mail: K.Sirlantzis@kent.ac.uk).

termed a *Judgement*. Judgements are usually written in the form  $p:P$  where the proof object  $p$  bears witness to the proposition  $P$ . Each logical connective in Constructive Type Theory is associated with four rules.

- Formation (syntax)
- Introduction
- Elimination
- Computation (simplification)

A simple example is the AND ( $\wedge$ ) rules.

$$\frac{A \text{ is a type} \quad B \text{ is a type}}{(A \wedge B) \text{ is a type}} (FORMATION)$$

$$\frac{a : A \quad b : B}{(a, b) : A \wedge B} (INTRO)$$

$$\frac{p : (A \wedge B)}{snd \ p : B} (ELIM)$$

$$\frac{p : (A \wedge B)}{fst \ p : A} (ELIM)$$

$$fst \ (a, b) \rightarrow a (COMP)$$

$$snd \ (a, b) \rightarrow b (COMP)$$

Difficulties in the practical application of Constructive Type Theory centre around the derivation of the proof of a proposition and the practical evaluation of the proof (although the proof represents an algorithm, it may not represent an efficient algorithm). The prime idea of this paper is to exploit learning algorithms derived from the field of Artificial Neural Networks to ease the burden on the programmer in deriving the required solution and hence allow for the practical exploitation of the logic.

### III. ARTIFICIAL NEURAL NETWORKS (ANN)

Artificial Neural Networks are employed today in a wide variety of application areas. These range from pattern [2] and character recognition [4] to financial time series forecasting [1], which call for a diverse set of different data representations ranging from binary values to values from the Clifford Algebra domain [10]. Faced with this unlimited diversity, those working in the field realised over the years the need for a unifying framework for a concise description of the various forms of neural networks and moreover, if possible, a formal specification system for them. As a result a number of systems were developed. One of the most notable was the one of the Neural Networks Council of the IEEE [9] for a canonical description of neural nets, the specification language and graphical interface developed under the NEUFODI European Project [3] and the mathematical model developed by L. Smith [12]. However these works are either descriptive or they do not provide a working prototype able to address real world problems, so

that the results can be compared against the specifications given.

Artificial Neural Networks present a powerful tool for the analysis of complex systems. However, ANN implementations are not usually amenable to formal analysis and verification of correct performance. This makes them inefficient tools for deriving algorithms meeting a given performance specification. Difficulties in their practical application centre around the derivation of suitable *weights* associated with the neural connections. On the other hand, Constructive Type Theory (CTT) offers the rigorous base of a formal logic and at the same time a formal specification tool which is capable of providing algorithmic solutions in the form of proofs of the propositions asserted. The purpose of the work reported here is to investigate the potential benefits of merging the areas of Artificial Neural Networks and Constructive Type Theory in a single framework in order to address the difficulties arising in the practical application of each one of them separately [5]. More specifically, this work aimed:

- To enhance the current state of neural networks technology by generalising the concepts of 'weight' and 'activation function' to assume values taken from the domain of Constructive Type Theory.
- To show the generality of this system by implementing an existing Neural Network architecture as a special cases of it.
- To illustrate the practical applicability of the system by developing an application to produce solutions to address a real world problem.

### IV. THE THEORETICAL FRAMEWORK

The structure of the prototype system introduced in this paper retains the abstract structure of a neural network as a directed graph which was described above. However, unlike the conventional neural architectures in which the weights are represented, for example, as real numbers, in the prototype they are expressed as *logical implications*  $A \rightarrow B$ , where  $A$  and  $B$  are datatypes. If  $A = \mathfrak{R}$  and  $B = \mathfrak{R}$ , where  $\mathfrak{R}$  represents the set of real numbers, then our prototype may, for example, represent the conventional real valued Multi-layer Perceptron (MLP). Its training algorithm (the generalised delta rule) can be viewed as modifying only the proof portion of the proposition/proof pair.

In fact, in our prototype, both weights and activation functions of the processing elements (nodes) comprising the network are expressed as logical implications. A node with two inputs,  $X$  and  $Y$  can then be represented as an implication of the type  $A \rightarrow B \rightarrow C$  where the weights associated with these two inputs can be  $X \rightarrow A$  and  $Y \rightarrow B$ . If all the types are numerical then the corresponding proof will represent a mathematical function and the prototype will describe a conventional neural network of some particular form.

The distributed nature of processing in neural networks helps to overcome the difficulties arising in the practical application of Constructive Type Theory, which centre around the *efficiency* and the *practical evaluation* of the derived algorithms.

Our implementation of the prototype is developed in the *Functional programming language Haskell* [14]. The advantage of using a functional language to realise our neural network prototype is threefold :

1. There exists a correspondence between Type Theoretic entities and functional programming objects
2. The expressions in a functional language preserve *referential transparency*. Furthermore, once proved correct they can be manipulated analogously to mathematical formulas. This allows the program to be considered a mathematical model.
3. Functional languages can be thought of as *executable formal specifications*.

In addition to these, our choice of Haskell for the implementation, which produces a compiled executable based on the C programming language, offered us a rather fast final program in comparison to the rather slowly running programs functional languages have been accused of producing

## V. THE PROTOTYPE

The purpose is to use and implement the algorithms derived in the Constructive Type Theory framework. The derivation process is then a mechanism for modifying definitions needed by the prototype. It is developed, subsequently, as a hierarchy of three levels of abstraction, which gives it the necessary flexibility to easily handle the variety of data forms and network types usually met in real world applications. Before presenting the definitions (in Haskell code, which in most cases is self-explanatory) a brief note on notation:

`::` indicates a type signature - that is its left operand has the type of its right operand -,  
`[A]` denotes a list of objects of type A and  
`->` denotes a logical implication (i.e. a function type).

Finally parentheses are used to indicate either grouping of results or clarify application of functions to a subset of their argument set (partial application of functions).

The first level of the prototype incorporates the definitions for the network itself and its components along with the functions that create them. In Haskell code these are expressed as follows:

```
data NeuralNet = NN{
    arch :: Net_Arch,
    fun  :: Net_Func,
    layer_nodes :: Layers}
  | ENN
where
type Net_Arch = Array Edge Weight
type Weight  = Maybe Weight_Function
```

```
type Weight_Function
    = Param -> Output -> Input
type Edge      = (Vertex, Vertex)
type Vertex    = Int
type Layers    = [[Vertex]]
type Net_Func  = [Node]
```

The above code defines a neural network to be an object which has an architecture (`Net_Arch`), a list of nodes (`Net_Func`), and a list of its vertices representing their arrangement in layers (`Layers`) (if this is applicable to the particular network). The architecture is represented as a 2-dimensional array of weights (`Weight`) indexed by the edges of the network.

A Neural Net is modelled as a record type with named fields so that each one of them can be recalled by name and possibly updated. The architecture is described as a 2-dimensional array with the elements representing the weights (logical implications - functions) in the cells corresponding to the connections of the vertices of the network. `Net_Func` is a list of `Node` type elements that express the functionality of the network. `Layers` is a list of lists of the (`Integer`) name tags of the nodes arranged in a layered configuration.

A node is defined as an object consisting of inputs (`Input`), output (`Output`), activation function (a logical implication from `Input` to `Output`), parameters associated with each one of its inputs, and error (`Delta`) to use during

```
data Node = Nd {
    inp  :: Inputs,
    outp :: Output,
    par  :: Params,
    activation_function ::
        Inputs -> Output,
    error :: Delta}
```

```
type Input = [Data]
type Inputs = [Input]
type Output = [Data]
type Param = [Data]
type Params = [Param]
type Delta = [Data]
```

the training. The corresponding code reads as shown below.

The second level of the abstraction implements functions which handle and update the network components. This level includes a general description of the learning process from the examples provided incorporating functions which define how to:

1. propagate “inputs” into a particular “neural\_net”, given a set of associated “parameters”, and produce the corresponding outputs,
2. evaluate the produced outputs with respect to “targets”, if required by the particular ANN model implemented, and
3. update the parameters of the node given a “learning\_rate”.

Training may now be defined as the composition (denoted by “.”) of the three functions (“propagate”, “evaluate” and “update”):

```
type Target = [Data]
train :: LParam -> [Input] -> [Target]
      -> [[Param]] -> NeuralNet ->
      NeuralNet
train learning_rate inputs targets
weights neural_net
  = ((update learning_rate) .
    (evaluate targets) .
    (propagate inputs
     parameters)) neural_net
```

The definition of the update function, which is used below to define two different ANN models, may be given by:-

```
type Lparam    = [Data]

update :: LParam -> NeuralNet ->
       NeuralNet
update lr nn    = nn {fun = set_params
                    nn ps}
```

where “lr” (of type “Lparam”) is the learning rate while “nn” indicates the current state of the neural network. Note that by implementing the neural network as a record type with named fields, the update operation is simplified. The right hand side of the equality in the “update” function definition in effect means that we replace the networks element called “fun” using a function “set\_params” which replaces the old parameter values stored in the network’s “nn” nodes with the new ones “ps”. These are calculated using the given learning parameter and the possibly the old weights’ values. In the following we will see that by appropriately defining “ps”- the list of the new (updated) parameters-we are able to implement a variety of ANN models with our prototype.

The third and final level of the prototype’s hierarchy comprises of the functions that form the operators on the various value types (e.g. Binary or Integer values). As soon as appropriate operators are defined at this level for any type of data, the prototype can be recompiled to produce an executable in order to process this kind of data. Thus, a 2-dimensional array of binary values representing a black and white image can be thought of as a single input to an input node and the parameters of the weight function associated with the network’s connections may themselves be multidimensional arrays. All that is needed is to define the appropriate function at the lower level of the hierarchy for the operations on these arrays.

Note that up to the second level of abstraction there is no requirement for any assumption about the specific type of the values that the data (“Data”) can take. To retain the flexibility of easily handling the variety of data forms and network types met in areas where integration with an existing system is needed, we chose to use a *polymorphic* datatype [14] defined in Haskell code as follows:

```
data Maybe A = Just A | Nothing
```

where A is a free variable representing any datatype. This in effect means that for any particular datatype “A” (e.g. Integer), which might be appropriate for a specific application, “Data” will either carry the information represented by it, taking a value of “Just A”, or carry no information at all, assuming the “Nothing” value. In the examples we present below, we chose “Data” to be “Maybe Float”, as the sensory device outputs, forming out sample data, are represented by real numbers. Then the “Data” associated with the input and output of every node in the network and the target values, if required in training, are defined as follows:

```
type Data = Maybe Float
```

This definition of the “Data” type offers itself to on-line (and hence possibly hardware) implementations because invalid sensory device output can be represented by the “Nothing” value so they would not convey any information (as indeed is the truth). We call this approach to data type definitions “domain restriction” because it can be used to restrict the domain of functions operating on the corresponding data types to the set of valid data values only.

## VI. A FEEDFORWARD NEURAL NETWORK

The first task in the derivation of a new network implementation algorithm is to use the rules of Constructive Type Theory to derive an algorithm which the programmer believes is a good approximation to the desired result. This is equivalent to a first prototype in a traditional Software Engineering task. The derivation of such an algorithm involves the application of the rules of Constructive Type Theory and a detailed description of the process is lengthy. Many examples are given in [6] and [13].

In order that this paper emphasises the way such a derived prototype may be merged with the paradigm of Artificial Neural Networks, we here assume that the derivation of a traditional Neural Network Architecture has been performed. We are not seeking to say that the derivation is trivial, but that such a derivation, once performed, may be used as the basis of an automated system capable of removing errors in the algorithm by means of given examples of the problem. In other words, the traditional testing and debugging phases of algorithm construction have been automated and, since the algorithm is expressed in a formal logical system, are amenable to formal mathematical analysis. The system may also be used to amend existing algorithms by “training” them in new example data.

The derivation of the network leads to the definition of the various functions required by the prototype. The first example is a simulation of a standard Multi-layer Perceptron (MLP). Firstly the node activation functions and the weight functions are derived so that they reflect the specific functional forms required by the model:

```
act_f :: NeuralNet -> Vertex ->
      (Inputs -> Output)
act_f nn = \i -> fnn_node_activ_func
           nn i
```

where “nn” is the neural network in its current state, “i” is a vertex number corresponding to the node whose activation function we define, and “x” is an input to this node. Then we have to derive “fnn\_node\_activ\_func” to assign to each node a specific, for example if the node is in the hidden layer the appropriate function could be data\_tanh (where tanh denotes the well known hyperbolic tangent function) defined as follows:

```
data_tanh Nothing = Nothing
data_tanh (Just y) = Just (tanh(y))
```

where y is an appropriate combination of the inputs of the node. Next we can define suitable weights (recall that the weights in our prototype are logical implications). The corresponding code is:

```
weight :: Weight_Function
weight = \par -> fnn_weight par

fnn_weight :: [Data] -> [Data] ->
           [Data]
fnn_weight = zipWith data_mult
```

where “zipWith f” is a function which when applied to two lists of objects, recursively applies “f” to the corresponding elements of the lists and returns the list of the results.

The final step is to set up an “update” function suitable for the model in hand. Recall that:

```
update lr nn = nn {
                fun = set_params nn ps}
```

and:

```
ps = learning_law lr nodes_2_update nn
```

where “ps” is the list of the new network parameters being updated according to a particular “learning\_law”, “lr” is the learning rate, “nodes\_2\_update” is the list of nodes whose parameters should be updated, and “nn” as usual is the neural network in its current (pre-updated) state. Then the last thing we need to define is the “nodes\_2\_update” list to be the list of all the non-input nodes in the network. Our FNN prototype is now completed. Below, we present one of the real world tasks on which this prototype has been employed

#### A. Forecasting the Dover Tides

The real world problem used to demonstrate the system arises from collaboration with the Dover Harbour Board. The Dover Harbour Board have been seeking a means to evaluate data relating to the current flows and tidal levels present within the harbour. Their main problem revolves around the complexity of the system comprising of the tides and the current flow mechanism which have proved difficult

to model and forecast. Conventional techniques have proved rather ineffective to date at producing satisfactory results and hence the application of neural networks has been considered.

The task presented here is to produce predictions of the tidal levels using a series of past measurements collected via a set of sensory devices within the harbour. The data set used consisted of hourly measurements of six variables considered affecting the tide dynamics. These variables are the tide level, the air and sea temperatures, the atmospheric pressure and wind speed and direction. The neural network used in this case had six one-dimensional input nodes (one for each one of the predictor variables), two hidden nodes and one output node. The activation functions selected for the hidden nodes are sigmoid nonlinearities (in particular the tanh function which has a range of (-1,1)). For the input and output nodes, the identity function was chosen.

The input variables were normalised to correspond to the effective range of the hidden nodes’ activation functions, while the weights were initialised for the training phase to random values uniformly distributed in (0,1). The training algorithm employed was the standard on-line version of Backpropagation of Errors [11]; here derived in the Constructive Type Theory framework.

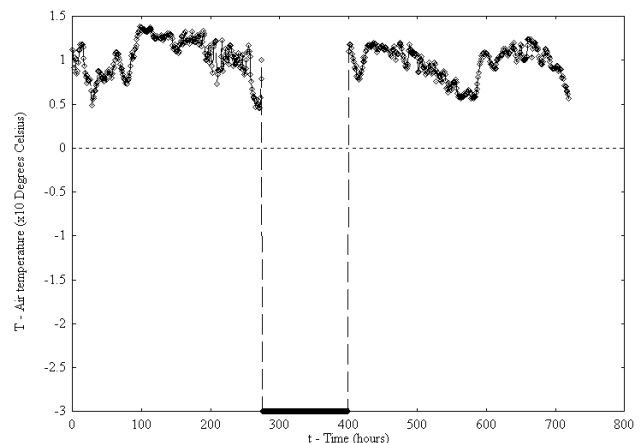


Figure 1: Air temperature measurements for November 1997.

The trained network was used to produce one-step-ahead predictions out-of-sample. That is a data set of the same configuration from a different month’s sample, namely November 1997, was used as inputs for our predictions. Figure 1 presents the 740 data points from this set for the air temperature measurements. Observe that between observations 280 to 400 the variable assumes values not consistent with the rest of the data set. In fact, these values represent a failure of the sensory system during the corresponding time period (can be thought of as corrupted data in general), a common problem in a practical real world environment.

Figure 2 illustrates the behaviour of the network implemented within our framework in comparison to one implemented in a conventional manner (for example using an imperative language like C). From this Figure, note that,

on the one hand, our implementation and the conventional network present errors with no significant differences for the time periods with no corrupted data (indices 230-280, and 400-430), and on the other hand, their performances deteriorate in a very similar way when data which contains corrupted measurements (indices 280-330) is presented to them.

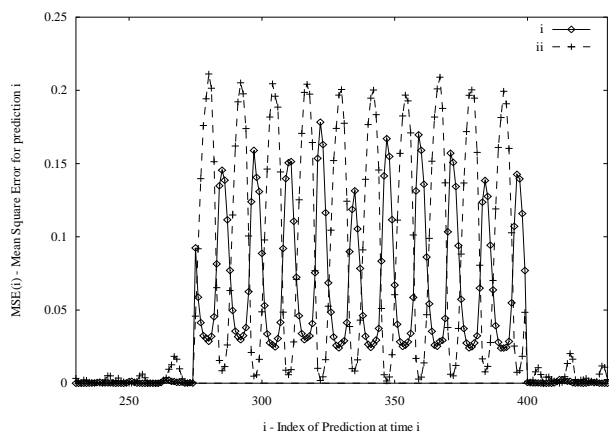


Figure 2: Prediction error; i) implementation of a MLP within our framework, without domain restriction, and using a training set with no corrupted air temperature values; ii) conventional implementation of MLP using the same training parameters and input data as in case i above

Conventionally, there are a number of ways to address the issue of corrupted data. We chose one of the most often used in practice to compare with our proposal of employing the “domain restriction” approach (see section V above) to tackle the problem.

Figure 3, shows the mean square error of the predictions produced using each of the above approaches for a period which contains both correct (indices 230-280) and corrupted data (indices 280-330). In order to have a basis for our comparisons we replot here (line with squares) the corresponding prediction error curve of the conventionally implemented MLP. The comparative advantage offered by our Prototype through the application of the “domain restriction” (line with diamonds in the Figure) can be easily verified.

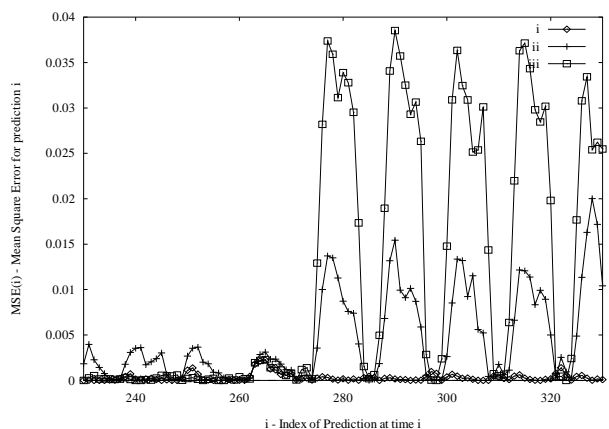


Figure 3: Prediction error; i) network using 6 inputs and domain restriction during the prediction phase and using a training set with no corrupted air temperature values; ii) network using 6 inputs without domain restriction but using a training set which includes corrupted data; iii) conventional

implementation of MLP using the same training parameters and input data as in case i above.

## VII. CONCLUSION

A mechanism has been introduced to illustrate the benefits of merging the areas of formal mathematics and artificial neural network. Such a technique is useful for problem domains which present difficulties in engineering an accurate solution but nevertheless possess a number of examples of valid results which may be employed as training data. This is a significant result in integrating formal logic with the field of Software Engineering.

## ACKNOWLEDGMENT

This research was supported by EPSRC grant No. GR/L78338. The authors wish to thank the Dover Harbour Board Hydrographic Service.

## REFERENCES

- [1] M.E Azoff, "Neural Network Time Series Forecasting of Financial Markets", John Wiley & Sons, 1994
- [2] C.M. Bishop, "Neural Networks for pattern recognition" Clarendon Press, Oxford, 1995
- [3] G Dorffner, H Wiklicky and E Prem, "Formal neural network specification and its implications for standardisation", *Computer Standards & Interfaces*, 1994, **16**, pp. 205-219
- [4] G. Howells, M.C. Fairhurst and F. Rahman, "An Exploration of a New Paradigm for Weightless RAM-based Neural Networks", *Connection Science: Vol. 12 No. 1*. March 2000.
- [5] G. Howells, K.Sirlantzis, "Improving Robotic System Robustness via a Generalised Formal Artificial Neural System" in *Proc. ECSIS Symposium on Learning and Adaptive Behaviour in Robotic Systems (LAB-RS 2008)* Edinburgh, 2008.
- [6] P Martin-Lof, "Constructive mathematics and computer programming". In C.A.R. Hoare (ed.) *Mathematical Logic and Programming Languages*, Prentice-Hall, 1985.
- [7] A Moran, D Sands and M Carlsson , "Erratic Fudgets: a semantic theory for an embedded coordination language", *Science of Computer Programming* 46 (1-2): 99-135 Jan-Feb 2003
- [8] Q.H. Nguyen, C Kirchner C and H Kirchner, "External rewriting for skeptical proof assistants", *Journal of Automated Reasoning* 29 (3-4): 309-336, 2002
- [9] M.L. Padgett, W.J. Karplus, S. Deiss and R. Shelton, "Computational Intelligence standards: Motivation, current activities, and progress", *Computer Standards & Interfaces*, 1994, **16**, pp. 185-203
- [10] A.F.R. Rahman, G. Howells and M.C. Fairhurst "A Multi-Expert Framework for Character Recognition: A Novel Application of Clifford Networks", *IEEE Transactions on Neural Networks: Vol. 12 No.1* January 2001
- [11] D.E. Rumelhart, G.E. Hinton and R.J. Williams, "Learning Representations by Back-Propagating Errors" , *Nature*, 1986) **323**, pp. 533
- [12] L. Smith, "A framework for neural net specification", *IEEE Trans. Soft. Eng.*, 1992, **18**, pp. 601-612
- [13] S. Thompson, "Type theory and functional programming" Addison-Wesley, 1991.
- [14] S. Thompson, "Haskell, the craft of functional programming" Addison-Wesley, 1999.