NSL: A NEURO–SYMBOLIC LANGUAGE FOR A NEURO–SYMBOLIC PROCESSOR (NSP)

ERNESTO BURATTINI and ANTONIO De FRANCESCO
Dipartimento di Scienze Fisiche, Università di Napoli “Federico II”
Napoli, Italy
cnb@na.infn.it, defranc@na.infn.it

MASSIMO De GREGORIO
Istituto di Oceanologia “Eduardo Caianiello”, CNR
Pezzuoli (NA), Italy
m.degregorio@ceb.na.cnr.it

A Neuro-Symbolic Language for monotonic and non-monotonic parallel logical inference by means of artificial neural networks (ANNs) is presented. Both the language and its compiler have been designed and implemented in order to translate the neural representation of a given problem into a VHDL software, which in turn can set devices such as FPGA. The result of this operation leads to an electronic circuit that we call NSP (Neuro-Symbolic Processor).

1. Introduction

In Aiello et al.\(^1\) has been introduced a neural model of forward chaining on propositional production rules (NFC, for short), based on a unified LCA (Localist Connectionist Architecture) approach\(^2\) of rule-based systems proposed by Burattini et al.\(^3\) in 1992. The NFC model suggests an approach to the parallel processing of rules. However, this approach becomes practically interesting since a hardware implementation of NFC has been provided – that is, NFCs have been transformed in what we call Neuro-Symbolic Processors (NSP, for short).

We present a Neuro-Symbolic Language (NSL) (formally introduced in Burattini et al.\(^4\)) that allows one to describe monotonic and non-monotonic parallel logical inferences by means of NFCs. For each program written in this language there exists the corresponding NFC neural network which is generated by a compiler program\(^5\) based on the Neuro-Symbolic approach introduced by Aiello et al.\(^6\)

The compiler, starting from a domain representation given in terms of production rules, generates the pseudo-code of the whole production system in terms of artificial neurons, and from the pseudo-code the corresponding VHDL code. The latter allows one to implement our NFCs on the chosen hardware.

The neuron model adopted here is the Weighted-Sum non-Linear Thresholded Element of McCulloch and Pitts.\(^7\) Its state transition function is defined as follows:

\[
p_i(t+1) = 1 \left( \sum_{j=1}^{k} (a_{i,j} \cdot p_i(t) - s_i) \right)
\]

where

\[
I[z] = \begin{cases} 
1 & \text{if } z > 0 \\
0 & \text{if } z \leq 0 
\end{cases}
\]

\(p_i(t)\) represents the state (1 or 0) of neuron \(p_i\) at time \(t\), \(a_{i,j}\) the coupling coefficient or weight between neurons \(p_i\) and \(p_j\), whose value may be positive (ex-

\(^{a}\)Copyright n. 001.109/D80.353 - 25/12/98.
citation) or negative (inhibition), $s_i$ the threshold of neuron $p_i$.

Inhibitions are used in the non-monotonic NFC model to implement parallel non-monotonic reasoning (in this case, inhibitions play a key role in asserting and retracting facts). In such NFCs, due to the use of inhibition, cyclic sequence of NFC states, namely reverberation, could be present. While the definitions of output and end-computation for non-reverberating networks give no problem and are easy to implement, we need to settle them for reverberating neural networks.

In the next sections we describe the NSL formalism and the non-monotonic NFC model. Furthermore, we introduce the definitions of output and end-computation for reverberating neural networks and the logical features of the non-monotonic NFC model. Finally, we show an NSL program that implements a simulation of the robot Myrmix.

2. Logical inference by ANN

As stated in Aiello et al., the two possible truth-values of a literal $p$ are represented by means of two distinguished neurons $p$ and $\neg p$: the first is activated if and only if the corresponding literal is supposed to be true, the second if and only if it is supposed to be false. Inactivity of these neurons means that we do not have information about the truth-value of the corresponding literal. A similar approach has been proposed in the past by von Neumann (Double Line Trick) to take into account non-monotonic behavior.

Using this kind of representation, we may assert that $p_i$ is true by exciting neuron $p_i$, that is false by exciting neuron $\neg p_i$; and if we wish to retract $p_i$ or $\neg p_i$, we send appropriate inhibition to the corresponding neuron. In the latter case of course we are entering the field of non-monotonic logic. The presence of both $p_i$ and $\neg p_i$ allows us also to check if an explicit contradiction arises on the basis of previous inferences.

In the asserting and retracting operations a meaningful role is played by inhibition. In fact, while a retract action from the external world means that some literal true until a given time no longer hold, a retract action resulting from an inferential process assumes the meaning that a literal previously believed as true can no longer be maintained in the light of some inference. The latter retract action is carried out by sending an inhibition to the neuron representing the literal in question.

Let us now introduce a suitable formalism for representing both monotonic and non-monotonic operators.

Let $P$ be a set of literals. We denote with $P^*$ and with $P_v$ respectively the conjunction and the disjunction of $P$ elements ($P_v$ will denote either $P^*$ or $P_v$). The basic operators we use to represent knowledge about a problem are classified into:

- monotonic operators IMPLY and ATLEAST;
- non-monotonic operators UNLESS and ATMOST.

Statement as

- "$p_i$ is true if $P_i$ is true"
- "$p_i$ is true if $P_v$ is true, unless $Q_{\neg v}$ is true"
- "$p_i$ is true if at least $h$ literals belonging to $P$ are true"
- "$p_i$ is true if at most $h$ literals belonging to $P$ are true"

are denoted respectively by:

- IMPLY($P_i$, $p_i$)
- UNLESS($P_{\neg v}$, $Q_{\neg v}$, $p_i$)
- ATLEAST($P_{\neg v}$, $h$, $p_i$)
- ATMOST($P_{\neg v}$, $h$, $p_i$).

These operators are represented in the NFC model by means of a net of neurons, using a Neuro-Symbolic approach formally discussed in Burattini et al. For instance, in fig. 1 is sketched the UNLESS($P_i$, $Q_v$, $p_i$) neural representation, where $P_i$ is the conjunction $p_1 \land \ldots \land p_n$ of $n$ literals and $Q_v$ the disjunction $q_1 \lor \ldots \lor q_m$ of $m$ literals.

Fig. 1. UNLESS($P_i$, $Q_v$, $p_i$) neural representation.
3. The Neuro-Symbolic Language

The NSL language allows one to write programs in order to generate NFC networks for representing and solving logical inference problems.

NSL is a context-free language and its grammar is written in the BNF (Backus-Naur) formalism. The language is formed by:

- a set of monotonic and non-monotonic operators;
- a set of iterative and conditional constructs (based on a Pascal-like syntax);
- a set of additional instructions (input/output instructions, assign instruction, ...).

Some formal expressions can be written as arguments of language operators, such as:

\[ p[S], p[S \ldots T], p[S, T] \]

where \( p \) is a literal, \( S \) and \( T \) two arithmetical expressions. Each of these expressions is translated by the NSL-compiler into the corresponding ones:

\[ p_1, p_1 \wedge \ldots \wedge p_t, p_1 t \]

where \( s \) and \( t \) are the results of evaluation of \( S \) and \( T \) expressions.

For instance, the instruction:

\[ \text{IMPLY}(a_1 \ldots i + 2) \wedge b[j], c[i, j]) \]

for \( i = 2 \) and \( j = 3 \), is translated into the equivalent one:

\[ \text{IMPLY}(a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge b_3, c_2.3). \]

The language takes advantage of constructs such as: FOR, IF THEN ELSE, WHILE DO and REPEAT UNTIL. For example, the following procedure:

\[ \text{FOR } (i, 1, 4) \]
\[ \text{BEGIN} \]
\[ \text{IMPLY}(a[i \ldots i], c[i+2]) \]
\[ \text{ATMEST}(c[i+2] \wedge d[i] \wedge e[i], 2, b[i]) \]
\[ \text{END} \]

produces the following code:

\[ \text{IMPLY}(a_1, c_2) \]
\[ \text{ATMEST}(c_0 \wedge d_1 \wedge e_1, 2, b_1) \]
\[ \text{IMPLY}(a_1 \wedge a_2, c_3) \]
\[ \text{ATMEST}(c_1 \wedge d_2 \wedge e_2, 2, b_2) \]
\[ \text{IMPLY}(a_1 \wedge a_2 \wedge a_3, c_6) \]
\[ \text{ATMEST}(c_2 \wedge d_3 \wedge e_3, 2, b_3) \]
\[ \text{IMPLY}(a_1 \wedge a_2 \wedge a_3 \wedge a_4, c_8) \]
\[ \text{ATMEST}(c_3 \wedge d_4 \wedge e_4, 2, b_4) \]

One can notice that the use of variables and of a FOR instruction allows one to represent the knowledge about the logical problem in a more compact way (see the example reported in section 7).

Another example shows the use of IF THEN ELSE construct:

\[ \text{READ}(i) \]
\[ \text{READ}(t) \]
\[ \text{IF } (i=1) \]
\[ \text{THEN} \]
\[ \text{BEGIN} \]
\[ \text{FOR } (j, 1, t) \]
\[ \text{IMPLY}(a[i \ldots j], c[j]) \]
\[ \text{END} \]
\[ \text{ELSE} \]
\[ \text{BEGIN} \]
\[ \text{FOR } (j, 1, t) \]
\[ \text{UNLESS}(a[j], b[j+1], c[j]) \]
\[ \text{END} \]

If \( i = 1 \) the compiler generates the code:

\[ \text{IMPLY}(a_1, c_1) \]
\[ \text{IMPLY}(a_1 \wedge a_2, c_2) \]
\[ \text{IMPLY}(a_1 \wedge a_2 \wedge a_3, c_3) \]
\[ \text{IMPLY}(a_1 \wedge a_2 \wedge a_3 \wedge a_4, c_4) \]

otherwise, the following one:

\[ \text{UNLESS}(a_1, b_2, c_1) \]
\[ \text{UNLESS}(a_2, b_3, c_2) \]
\[ \text{UNLESS}(a_3, b_4, c_3) \]
\[ \text{UNLESS}(a_4, b_5, c_4) \]

In the above example, both codes are generated for \( t = 4 \).
Usually, we can represent a logical problem just in terms of a set of operators. This means that if we change the value of a parameter, we are obliged to rewrite part of the operators. With an NSL program, thanks to the introduction of variables, we can approach the same logical problem in a more general way. In fact, the previous example emphasizes how the same NSL program generates different code with respect to external parameters given as input (READ(1), READ(2)).

We report here the remaining instructions of the NSL language:

- **ASSIGN(</var>, <expr>)** set the variable <var> to <expr> evaluation;
- **READ(</var>)** read <var> from the standard input stream;
- **WRITE(<message>)** write <message> to the standard output stream;
- **INPUT(I_1, ..., I_n)** set the possible inputs for the corresponding NFC network;
- **OUTPUT(O_1, ..., O_m)** set the possible outputs for the NFC network.

4. NFC networks for non-monotonic logical inference

The constructs and the instructions introduced in the previous section, allow one to translate any NSL program into the corresponding set of NSL operators (called NSLO version). The latter is formed only by monotonic and non-monotonic operators and by **INPUT** and **OUTPUT** instructions.

We have designed and implemented a **Neuro-Symbolic Compiler** for deriving automatically the NFC neural network, corresponding to any NSL program, into two different formats: NFC simulation code and NFC VHDL code. While the first is used to run the simulation of the NFC network on PCs, the second is used to appropriately set electronic devices (for instance FPGA – Field Programmable Gate Arrays).

The compiler is formed by three different modules (see fig. 2).

The **Syntactic Module** carries on a syntactic analysis of P. If no syntactic errors occur, the next module, called **Pre-Compiler Module** translates P into the corresponding NSLO version P0. At last, the **Compiler Module** derives the NFC network, which can be both simulated by means of Simulation Module, or implemented on a FPGA device.

![Fig. 2. Neuro-Symbolic compiler structure.](image)

We call the NFC network generated by the program P:

- **monotonic if P contains just monotonic operators**;
- **non-monotonic if P contains at least one non-monotonic operator**.

The non-monotonic NFC model implements a non-monotonic reasoning, that is, the introduction of new premises (literals), initially supposed true, could invalidate previously inferred conclusions.

Since the monotonic NFC model is widely described in Aiello et al.\(^7\), we focus our discussion only on the non-monotonic one.

Let P be the following NSLO program:

\[ > \text{UNLESS}(a, p, q) \]
\[ > \text{UNLESS}(a, q, p) \]
\[ > \text{IMPLI}(b, a). \]

In fig. 3 we report the non-monotonic NFC network corresponding to P (dashed lines indicate inhibitory connections).

The first layer (IN) accepts external inputs to the network and it is used to assert the set of initial premises supposed to be true (also called facts). It could be formed by as many neurons as different literals appearing in the P instructions. We can reduce their number by means of **INPUT** instruction, that is,
only the neurons corresponding to the literals of the INPUT instruction will be present in the IN layer.

![Diagram](image)

Fig. 3. Non-monotonic NFC network for \( P \).

The second layer (DB) is a database formed by as many neurons as in the layer IN, and it stores the premises asserted in the IN layer thanks to self-excitation links.

The third layer (KB) codifies the entire knowledge base and the inferential process (see fig. 4). This layer is based on neural representation of monotonic and non-monotonic operators, as discussed in §2 (see fig. 1 for the UNLESS neural representation).

![Diagram](image)

Fig. 4. KB layer.

At last, the OUT layer represents the set of possible conclusions (as for the neurons belonging to the IN layer, the number of output neurons is given by the OUTPUT instruction). The END activation indicates that the end of computation has been reached.

The End-computation & Output network is formed by two different sub-networks. The End-computation sub-network controls the completion of the inferential process, while the Output sub-network selects the results of the computation. Both sub-networks are designed and perform according to the definitions given in the next section.

In the previous example (see fig. 3), exciting the neuron \( b \) of the IN layer one obtains the KB neurons temporal evolution shown in fig. 5 (where \( \{a, b, p, q\} \) represent states of \( KB \) layer).

![Diagram](image)

Fig. 5. KB layer temporal evolution.

One can notice that both neurons \( a \) and \( b \) keep firing indefinitely, while \( p \) and \( q \) neurons keep oscillating, that is, they pass from quiescent to exciting state, and vice versa indefinitely (reverberating network). Intuitively, we could say that facts \( a \) and \( b \) are the conclusions of the inferential process, while \( p \) and \( q \) are oscillating literals for which we are not able to determine their truth-value. In order to define the set of conclusions in case of reverberating networks, we need to define the output of such networks. In the next section, we propose a suitable End-computation and Output definitions.

5. Transient and reverberation in non-monotonic NFC

In the temporal evolution of a non-monotonic NFC network, whose states depend only on the immediately previous one, it can be proved¹² that after an eventual transient phase a cyclic behavior may occur (reverberation).

![Diagram](image)

Fig. 6. Temporal evolution of a generic non-monotonic NFC.

Let \( I \) the input of a non-monotonic NFC network and \( U(t) \) the generic network state at time \( t \) (that is the set of \( KB \) nodes firing at time \( t \)). We denote with \( U(0), \ldots, U(\tau - 1) \) the transient phase and with the oscillating states \( U(\tau), \ldots, U(\tau + L) \) the reverberation. \( L \) is called reverberation period of the network for input \( I \).
Temporal evolution of a reverberating non-monotonic NFC networks is reported in fig. 6.

If \( L > 0 \), an infinite cyclic sequences of oscillating states is present. If \( L = 0 \) the network is called **reverberation free or non oscillating**, and the unique state \( U(\tau) \) belonging to the reverberation is called **stable state**.

Given an input \( f \), for non oscillating networks \( (L = 0) \) the output, that is formally the set of conclusions, is equal to the stable state; computation ends at time \( \tau \), that is, when stable state is reached.

For reverberating NFC networks \( (L > 0) \) we have to introduce a suitable output and end-computation definitions:

- **the output** is the intersection \( U(\tau) \cap \ldots \cap U(\tau + L) \) of oscillating states, namely, the set of \( KB \) neurons always active in \( L \);
- **network computation ends** when all oscillating states have been **labelled**. To label a state we must wait that network evolves twice into it. This is the only way to distinguish transient states by oscillating ones: in fact, a network evolves only one time in the former and more times in the latter.

From the previous definitions, the output of the network for the example reported in fig. 5 is \( \{ a, b \} \) (that is: \( \{ a, b \} \cap \{ a, b, p, q \} \)).

In a reverberating NFC network there is a reverberating cyclic sequence of oscillating states, and we can consider a computation ended when all these states have been labelled. Then, we do not consider oscillating literals as belonging to the set of conclusions (they are asserted and retracted every reverberation period). We accept as conclusions of the logical inferential process only those literals belonging to each single reverberation state.

### 6. NFC model and logic program

Let us remind some definitions.

A propositional normal logic program \( P \) is a collection of clauses of the form

\[
a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m
\]

where \( m, n \geq 0 \) (the literal \( a \) is called conclusion and the literals \( b_1, \ldots, b_n, c_1, \ldots, c_m \) are called premises).

A program \( P \) is **definite** (or **negation-free**) if contains only clauses of the form \( a \leftarrow b_1, \ldots, b_n \) (Horn clauses).

An abstract interpreter of logic programs is based on uniﬁcation algorithm and resolution method, formulated by Robinson in 1965.

A normal logic program interpreter is based on the NFR (negation as failure rule) interpretation of \( \neg \) symbol, that is only a partial form of logical negation.

Moreover, a premises-free clause \( a \leftarrow \) is called **fact**, while the remaining clauses are called **rules**.

Let \( a \leftarrow b_1, \ldots, b_n \) be a rule of a deﬁnite logic program \( P \). We translate it in our language with an **IMPLY operator**:

\[
IMPLY(b_1 \land \ldots \land b_n, a).
\]

In this way, we take into account the rules in the \( KB \) layer of the monotonic NFC model corresponding to \( P \), while the facts are asserted by exciting nodes of the \( IN \) layer.

Since the rules are processed and executed all at once, we can assert that:

- the NFC model is a massively parallel abstract interpreter of deﬁnite logic programs. (In de Francesco\(^\text{1}\) has been proved the correctness and completeness features of the monotonic NFC model);
- NFC network structure depends exclusively on the set of logical rules (it does not depend on the set of facts);
- NFC network simulates a class of logic definite programs for any instance of the initial facts.

Given an instance of the set of initial facts as input to the network then there exists a one-to-one correspondence between logic program and NFC network.

In a similar way, a rule with negation, such as \( a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m \), is written as follows:

\[
UNLESS(b_1 \land \ldots \land b_n, c_1 \lor \ldots \lor c_m, a).
\]

We observe that in the neural representation of **UNLESS** operator the inhibition is used to implement a neural form of logical negation.

Relationships between logic programs and non-monotonic NFC networks are still an open problem.
7. An example of a robot simulation

In order to give an example of an actual non-monotonic NFC application, let us consider a simplified version of the robot Myrmic¹⁵. The task of Myrmic is to find food and to eat it. In this example the robot collects food by grasping, and then eats and digests it. By eating we mean that whenever Myrmic detect food, it turns toward the target, stops in front of it, and eats the food. After a certain amount of time, the robot turns away and starts moving straight ahead again. Myrmic takes a certain time to digest its food, and during the digestion period it simply avoids all objects it encounters. As soon as the food has been entirely digested, it engages in looking for food again.

The robot has 5 sensors:

- **OFS**: obstacle front sensor to check whether there is an obstacle in the robot’s way;
- **OBS**: obstacle back sensor to check whether there is an obstacle back the robot;
- **OLS**: obstacle left sensor to check whether there is an obstacle left the robot’s way;
- **ORS**: obstacle right sensor to check whether there is an obstacle right the robot’s way;
- **FS**: food sensor to detect if in the front there is some food.

![Diagram of robot architecture](image)

The robot control architecture is performed by the subsumption strategy applied to three main layers: Safe Forward, Avoid Obstacles, Collect Food.

**Safe Forward** – this layer causes the robot to move straight ahead or back making sure that it does not collide with obstacles. These actions are performed unless the robot is turning or has detected food or is eating.

**Avoid Obstacles** – this layer causes the robot turns right or left according to the presence or absence of obstacles on right or left. This action is performed unless the robot has detected food or is eating.

**Collect Food** – this layer causes the robot looks for food. Whenever the food has been detected, unless the robot is digesting, it tries to centre the food, after that the robot eats it and then it digests.

In fig. 7 is reported a diagram of the layers and modules representing the robot architecture. Each layer sends information to some other layers and inhibition to suppress their activities when necessary.

We may represents the Myrmic architecture by using our approach.

The three layers can be represented by the following operators:

**Safe Forward layer**

\[ \text{UNLESS}(\text{OFS}, (\text{turn} \lor \text{detect} \lor \text{eat}), \text{m.back}) \] – move backward unless is turning or has detected food or is eating.

\[ \text{UNLESS}(\text{OBS}, (\text{turn} \lor \text{detect} \lor \text{eat}), \text{m.forward}) \] – move forward unless is turning or has detected food or is eating.

**Avoid Obstacles layer**

\[ \text{UNLESS}(\text{OLS}, (\text{detect} \lor \text{eat}), \text{turn.right}) \] – turn right unless has detected food or is eating.

\[ \text{UNLESS}(\text{ORS}, (\text{detect} \lor \text{eat}), \text{turn.left}) \] – turn left unless has detected food or is eating.

**Collect Food layer**

\[ \text{UNLESS}(\text{FS}, \text{digest}, \text{detect}) \] – detect food unless is digesting.

The other main modules in the layers are represented by:

\[ \text{IMPLY} (\text{detect}, \text{centre.food}) \] if detect food then centre food.

\[ \text{IMPLY} (\text{centre.food}, \text{grasp}) \] if centre food then grasp.

\[ \text{IMPLY} (\text{grasp}, \text{eat}) \] if grasp then eat.
IMPPLY(eat, digest)  
if eat then digest. (The digestion ends after a fixed amount of time.)

The neural network architecture for this simplified version of the robot Myrmix is reported in fig. 8.

![Diagram of Myrmix architecture]

Fig. 8. NFC Myrmix architecture.

The following program allows us to run a simulation with an arbitrary number of robots (N.robots):

```plaintext
> WRITE('Insert number of robots: ')  
> READ(N.robots)  
> FOR (i, 1, N.robots)  
> BEGIN  
>   UNLESS(OFS[i],  
>      (turn[i] V detect[i] V eat[i]),  
>      m.back[i])  
>   UNLESS(OLS[i],  
>      (turn[i] V detect[i] V eat[i]),  
>      m.forward[i])  
>   UNLESS(ORS[i],  
>      (detect[i] V eat[i]),  
>      turn.right[i])  
>   UNLESS(OLS[i],  
>      (detect[i] V eat[i]),  
>      turn.left[i])  
>   UNLESS(FS[i],  
>      (detect[i] V eat[i]),  
>      digest[i])  
>   IMPLY(eat[i], digest[i])  
> END
```

8. Concluding remarks

The idea of designing computers using threshold elements or neurons is as old as the McCulloch and Pitts seminal paper. Minsky in 1956 and later in 1967 reported on the possibility of implementing most operations performed by a von Neumann machine using McCulloch and Pitts neurons. Minsky claimed that:

*As the control over fabrication methods improves, we can expect the more delicate “threshold-logic” kind of circuit to play a large role.*

Minsky's expectations about technological improvements are partially met by the technology now available on the devices market: the FPGA. These processors enable one to implement logical structures like those reported in the present paper.

The NSL language and its compiler (that translates NSL programs to NFC networks, and transforms NFC networks into VHDL code), allow one to easily express logical problems and to translate them into the corresponding VHDL code to set electronic devices like, for instance, FPGAs. Each of these devices is reprogrammable and, should some of the NSL instructions be changed for some reason, the same reprogrammable device is ready to be used again, after that the modified NSL program has been compiled again.

In conclusion, we may claim that on FPGA devices we can effectively implement a massively parallel interpreter of logic programs.

References


