Towards reasoning about the past in neural-symbolic systems

Rafael V. Borges and **Luís C. Lamb**
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre RS, 91501-970, Brazil
rvborges@inf.ufrgs.br; LuisLamb@acm.org

Artur S. d'Avila Garcez
Department of Computing
City University London
London EC1V 0HB, UK
aag@soi.city.ac.uk

Abstract

Reasoning about the past is of fundamental importance in several applications in computer science and artificial intelligence, including reactive systems and planning. In this paper we propose efficient temporal knowledge representation algorithms to reason about and implement past time logical operators in neural-symbolic systems. We do so by extending models of the Connectionist Inductive Learning and Logic Programming System with past operators. This contributes towards integrated learning and reasoning systems considering temporal aspects. We validate the effectiveness of our approach by means of case studies.

1 Introduction

Neural-symbolic systems address the need of bridging the gap between the symbolic and connectionist paradigms of artificial intelligence [3]. The ability to reason about and learn with past information is a relevant aspect of intelligent behaviour [5]. In systems as [2] information about the past is fundamental in the decision making process of reactive systems. Past time operators have also been shown applicable in the study of knowledge-based programs [9]. Some connectionist systems effect temporal reasoning and learning based on past information, e.g. [8; 12].

On the other hand, a systematic neural-symbolic method to treat non-classical logics has only recently been outlined [4; 5; 6]. The *Connectionist Inductive Learning and Logic Programming System* (CILP) [7] is a neural-symbolic system based on the translation of logic programs into a connectionist architecture with sound semantics and good performance in real world applications. *Connectionist Modal Logics (CML)* proposed in [5; 6] have combined the strengths of non-classical logics and neural networks by offering integrated reasoning and learning within the same computational model. In CML one is able to compute and learn expressive non-classical logics, including modal, temporal, and intuitionistic reasoning [5; 6]. This work follows the same line of research. In particular, the approach presented in this paper contributes towards networks' complexity issues. We present an algorithm capable of representing and reasoning about temporal information without the need of replicating the structure of a given network. We introduce *Sequential Connectionist Temporal Logic (SCTL)*, an approach that deals with past temporal knowledge in CILP and CML-based models, proposing improvements to their translation algorithms. Our work builds upon temporal logic programs, using past operators only, and proposes algorithms to effect the translation of temporal logic programs into a connectionist architecture.

Section 2 contains background information on logics; Section 3 describes the algorithm that translates temporal programs into neural networks; Section 4 analyses the behaviour of the networks when computing a program; Section 5 concludes and points out directions for further research.

2 Preliminaries

In this section we introduce basic definitions of logic programming and temporal reasoning used in this paper.

Definition 2.1 *An* atom *A is a propositional variable; a* literal *is an atom A or a negation of an atom ($\sim A$). A* temporal atom *is recursively defined by the following: (i) Every atom A is a temporal atom; (ii) if α and β are temporal atoms, then $\bullet\alpha, \blacksquare\alpha, \blacklozenge\alpha$ and $\alpha \mathbb{S} \beta$ are temporal atoms. We define a* temporal literal *λ as a temporal atom or its negation. A* clause *is an implication of the form $\alpha \leftarrow \lambda_1, \lambda_2, ..., \lambda_n$ with $n \geq 0$, where α is a temporal atom and λ_i, for $1 \leq i \leq n$, are literals. In the remainder of the paper, we shall refer to the* temporal atoms *simply as* atoms. *A program is a set of clauses.*

Definition 2.2 *[1] Let $B_{\mathcal{P}}$ denote the Herbrand base of a temporal logic program $\mathcal{P}$, i.e., the set of temporal atoms occurring in $\mathcal{P}$. A* level mapping *for a program $\mathcal{P}$ is a function $|\ |\ : B_{\mathcal{P}} \rightarrow \mathcal{N}$ of ground atoms to natural numbers. For $\alpha \in B_{\mathcal{P}}, |\alpha|$ is called the level of α and $|\sim \alpha| = |\alpha|$. A sequential temporal logic program $\mathcal{P}$ is* acceptable *w.r.t a level mapping $|\ |$ and a model $\boldsymbol{m}$ of $\mathcal{P}$ if, for every clause $\alpha \leftarrow \lambda_1, ..., \lambda_k$ in $\mathcal{P}$, and $1 \leq i \leq k$, the following implication holds: **if** $\boldsymbol{m} \models \lambda_1 \wedge ... \wedge \lambda_{i-1}$ **then** $|\alpha| > |\lambda_i|$. A program $\mathcal{P}$ is acceptable if it acceptable w.r.t. some level mapping and some model. A propositional program $\mathcal{P}$ is* acyclic *w.r.t a level mapping $|\ |$ if, for every clause $\alpha \leftarrow \lambda_1, ..., \lambda_k$ in $\mathcal{P}$, $|\alpha| > |\lambda_i|$ for $1 \leq i \leq k$. A program $\mathcal{P}$ is* acyclic *if it is acyclic w.r.t some level mapping.*

Proposition 2.3 *[1] Every acyclic program $\mathcal{P}$ is also acceptable.*

The semantics of a logic program can be defined by means of the fixed point of the immediate consequence operator [10]. For acceptable programs, this operator converges to a unique fixed point. Since this operator serves as basis for the translation algorithm, we shall define these notions for *sequential temporal logic programs*.

Definition 2.4 *The immediate consequence operator $T^t_{\mathcal{P}}$ of a temporal program $\mathcal{P}$ at a time $t \geq 1$ maps an interpretation $I^t_{\mathcal{P}}$ of the temporal atoms of the program to a new interpretation of these atoms, where $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is true if there is a clause of the form $\alpha \leftarrow \lambda_1, ..., \lambda_n$ and $I^t_{\mathcal{P}}(\lambda_i)$ is true for all $1 \leq i \leq n$.*

When $t = 1$ (the initial point), past information is not considered, and an atom is considered true iff it is head of a clause of the form $\alpha \leftarrow \lambda_1, ..., \lambda_n$. For the other time points, other semantic rules, considering past information, must be taken into consideration to compute the $T^t_{\mathcal{P}}$ operator for $t > 1$. We shall define these new rules in the sequel.

2.1 Past time operators

The basic operator that makes reference to the past is the *previous time* operator $\bullet$. It refers to information at the immediately prior point in the time line. A new semantic rule must be considered to deal with this operator in the computation of the immediate consequence operator:

Definition 2.5 *The immediate consequence operator of a program $\mathcal{P}$ maps an interpretation $I^t_{\mathcal{P}}$ at time t to a new interpretation assigning true to a atom of the form $\bullet\alpha$ if α is true at time $t-1$, i.e. $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\bullet\alpha) =$ **true** if $\mathcal{F}^{t-1}_{\mathcal{P}}(\alpha) =$ **true**, where $\mathcal{F}^{t-1}_{\mathcal{P}}$ is the fixed point of program $\mathcal{P}$ at $t-1$.*

The semantics of the remaining past operators may be recursively defined through the use of the previous time operator. These definitions allow a simple representation of these operators in a connectionist setting. We also note that we assume a non-strict definition of past time, i.e. we consider the present time. The $\blacksquare$ operator (*always in the past*) denotes that an atom is true at every time point in the past. It is recursively defined as $\blacksquare\alpha \leftrightarrow \alpha \wedge \bullet\blacksquare\alpha$. The $\blacklozenge$ operator (*sometime in the past*) is the dual of the $\blacksquare$ operator. It denotes that an atom is true in some previous time point. It is recursively defined as $\blacklozenge\alpha \leftrightarrow \alpha \vee \bullet\blacklozenge\alpha$. The $\mathbb{S}$ binary operator (*since*) represents that an atom α has been true since the last occurrence of another atom β. It is recursively defined as $\alpha \,\mathbb{S}\, \beta \leftrightarrow \beta \vee (\alpha \wedge \bullet(\alpha \,\mathbb{S}\, \beta))$ At time $t = 1$, $\bullet(\alpha \,\mathbb{S}\, \beta)$ and $\bullet\blacklozenge\alpha$ are assigned *false*, and $\bullet\blacksquare\alpha$ is assigned *true*. At the remaining points, $\bullet\alpha$ is interpreted w.r.t. the fixed point of the previous time; i.e. the fixed point at a time t must be calculated before the execution of the $T^{t+1}_{\mathcal{P}}$ operator. Therefore, the following rules must be considered in the computation of the fixed point of a program:

Definition 2.6 *The immediate consequence operator of a program $\mathcal{P}$ maps an interpretation $I^t_{\mathcal{P}}$ at time t to a new interpretation assigning true to atoms as follows:*

*1. $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\blacksquare\alpha) =$ **true** if $I^t_{\mathcal{P}}(\alpha) =$ **true** and $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacksquare\alpha) =$ **true**;*

*2. $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\blacklozenge\alpha) =$ **true** if $I^t_{\mathcal{P}}(\alpha) =$ **true** or $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacklozenge\alpha) =$ **true**;*

*3. $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha \,\mathbb{S}\, \beta) =$ **true** if $I^t_{\mathcal{P}}(\beta) =$ **true**;*

*4. $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha \,\mathbb{S}\, \beta) =$ **true** if $I^t_{\mathcal{P}}(\alpha) =$ **true** and $\mathcal{F}^{t-1}_{\mathcal{P}}(\alpha \,\mathbb{S}\, \beta) =$ **true**; where, by definition, $\mathcal{F}^0_{\mathcal{P}}(\blacksquare\alpha) =$ **true**, $\mathcal{F}^0_{\mathcal{P}}(\blacklozenge\alpha) =$ **false** and $\mathcal{F}^0_{\mathcal{P}}(\alpha \,\mathbb{S}\, \beta) =$ **false**.*

A program $\mathcal{P}$ in which the *previous time operator* $\bullet$ is the only temporal operator is called $\bullet$-based program.

Definition 2.7 *The immediate consequence operator $T^T_{\mathcal{P}}(I^t_{\mathcal{P}})$ of a program $\mathcal{P}$ maps any interpretation $I^t_{\mathcal{P}}$ at a time t to a new interpretation assigning true to an atom α iff α follows one of the rules in definitions 2.4, 2.5, 2.6. The restricted operator $\bullet T^t_{\mathcal{P}}(I^t_{\mathcal{P}})$ maps $I^t_{\mathcal{P}}$ to a new interpretation assigning true to an atom α iff α is constructed as in definitions 2.4, 2.5.*

3 Translation Algorithm

In this section we describe the algorithm that translates a temporal logic program into a recurrent neural network.

3.1 Temporal Conversion

The first step of the algorithm consists in converting a temporal logic program $\mathcal{P}$ into a new $\bullet$-based program $\mathcal{P}_1$. This step is executed in order to adapt the program to the connectionist architecture that shall be used to represent the semantics of the program, since this architecture allows only the computation of the previous time operator. As described in the previous section, each temporal operator can be represented as a recursive function of the fixed point in the previous time. This can be represented by the insertion of clauses in the system, as in Fig. 1. Note that in the new program the temporal atoms where the operator is different from the previous time operator are considered as simple atoms, i.e. the semantics of the remaining operators is not considered.

```
TempConversion(P)
    foreach γ ∈ Atoms(P) do
        if γ = ■α then
            AddClause(P, ■α ← ●■α, α)
        end
        if γ = ♦α then
            AddClause(P, ♦α ← ●♦α)
            AddClause(P, ♦α ← α)
        end
        if γ = α S β then
            AddClause(P, α S β ← β)
            AddClause(P, α S β ← ●(α S β), α)
        end
    end
    return (P)
end
```

Figure 1: First step of the translation

Lemma 3.1 *The application of the $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})$ operator of a temporal program $\mathcal{P}$ over an interpretation $I^t_{\mathcal{P}}$ and the application of the $\bullet T^t_{\mathcal{P}}(I^t_{\mathcal{P}})$ operator of a $\bullet$-based program $\mathcal{P}_1 = TempConversion(\mathcal{P})$ over the same interpretation assigns the same valuation for all atoms $\alpha \in B_{\mathcal{P}}$.*

Proof(sketch): The rules in definitions 2.4 and 2.5 define both $T^t_{\mathcal{P}}(I^t_{\mathcal{P}})$ and $\bullet T^t_{\mathcal{P}}(I^t_{\mathcal{P}})$, so the result of the immediate consequence operator application for atoms that follow these two definitions is the same. The clauses inserted by the algorithm and Def. 2.5 are enough to ensure the computation of $T^t_{\mathcal{P}}$ according the recursive definition 2.6, without inserting any false assignment. □

Example 3.2 *Suppose the following about the development of a system: "A program had no errors before the last modification. Since the last modification, it is producing an undesired result. Therefore, the last modification has to be corrected". Table 1 shows two examples of temporal sequences, illustrating the assignment of truth values to the atoms. E represents the existence of an error in the program, L, represents that the last modification is being made, and Corr, denotes that a correction must be made over the last modification.*

Case a:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
E	F	F	F	F	T	T	T
L	F	F	T	T	F	F	F
$Corr$	F	F	F	F	T	T	T

Case b:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
E	F	T	T	T	T	T	T
L	F	F	T	T	T	F	F
$Corr$	F	F	F	F	F	F	F

Table 1: Temporal sequences describing the example

Table 2 shows a possible description of this problem as a temporal logic program. Tmp is an atom which is *true* if the last modification is made after a time point where the program has no error. The first two lines show a program representing this problem, and the remaining lines contain the clauses inserted in the program by the function $TempConversion$. Note that two temporal atoms in the program have been translated. Atom $E \mathbb{S} L$ is represented by clauses 3 and 4, and $\blacklozenge Tmp$ is represented by clauses 5 and 6.

1	$Corr \leftarrow E, E \mathbb{S} L, \blacklozenge Tmp$
2	$Tmp \leftarrow L, \sim \bullet E$
3	$E \mathbb{S} L \leftarrow L$
4	$E \mathbb{S} L \leftarrow \bullet(E \mathbb{S} L), E$
5	$\blacklozenge Tmp \leftarrow Tmp$
6	$\blacklozenge Tmp \leftarrow \bullet\blacklozenge Tmp$

Table 2: Example of the first step of the algorithm

3.2 CILP's Algorithm Application

The main step of the translation consists in the application of CILP's algorithm[7]. The process consists in a localist representation of the program in a three-layered neural network, where each atom is represented by neurons in the input and the output layer and each clause is represented by an hidden neuron. Every temporal atom will be treated by the translation as classical atoms are treated in the original algorithm. CILP's algorithm is described in Fig. 2.

In Fig 2 $max_{\mathcal{P}}(k,\mu)$ is the maximum value among the number of literals in a clause and the number clauses with the same head in program $\mathcal{P}$, where k is the number of literals in the body of a clause, μ is the number of clauses with the same head; A_{min} is the minimum activation value for a neuron to be considered active (or true). Neurons in the input layer are labelled in_α; neurons in the output layer are labelled

ExecuteCILP($\mathcal{P}$)
  Define $\frac{max_{\mathcal{P}}(k,\mu)-1}{max_{\mathcal{P}}(k,\mu)+1} \leq A_{min} < 1$
  Define $W \geq \frac{ln(1+A_{min})-ln(1-A_{min})}{max_{\mathcal{P}}(k,\mu)(A_{min}-1)+A_{min}+1} \cdot \frac{2}{\beta}$
  foreach $C_l \in Clauses(\mathcal{P})$ **do**
    $AddHiddenNeuron(\mathcal{N}, h_l)$
    foreach $\alpha \in body(C_l)$ **do**
      if $in_\alpha \notin Neurons(\mathcal{N})$ **then**
        $AddInputNeuron(\mathcal{N}, in_\alpha)$
        $ActivationFunction(in_\alpha) \leftarrow g(x)$
      end
      $AddLink(\mathcal{N}, in_\alpha, h_l, W)$
    end
    foreach $\sim \alpha \in body(C_l)$ **do**
      if $in_\alpha \notin Neurons(\mathcal{N})$ **then**
        $AddInputNeuron(\mathcal{N}, in_\alpha)$
        $ActivationFunction(in_\alpha) \leftarrow g(x)$
      end
      $AddLink(\mathcal{N}, in_\alpha, h_l, -W)$
    end
    $\alpha \leftarrow head(C_l)$
    if $out_\alpha \notin Neurons(\mathcal{N})$ **then**
      $AddOutputNeuron(\mathcal{N}, out_\alpha)$
    end
    $AddLink(\mathcal{N}, h_l, out_\alpha, W)$
    $Threshold(h_l) \leftarrow \frac{(1+A_{min})(k_l-1)}{2} W$
    $Threshold(out_\alpha) \leftarrow \frac{(1+A_{min})(1-\mu_l)}{2} W$
    $ActivationFunction(h_l) \leftarrow h(x)$
    $ActivationFunction(out_\alpha) \leftarrow h(x)$
  end
  foreach $\alpha \in atoms(\mathcal{P})$ **do**
    if $(in_\alpha \in neurons(\mathcal{N})) \wedge (out_\alpha \in neurons(\mathcal{N}))$ **then**
      $AddLink(\mathcal{N}, out_\alpha, in_\alpha, 1)$
    end
  end
  return $\mathcal{N}$
end

Figure 2: CILP's Translation Algorithm

out_α where α is the atom represented by these neurons. h_i are hidden neurons representing each clause of the program. $AddLink(\mathcal{N}, source, target, W)$ denotes the insertion of a link from a neuron *source* to a neuron *target* in a network $\mathcal{N}$, with weight W.

Lemma 3.3 *([7])For each classic logic program $\mathcal{P}$, there exists an artificial 3-layered neural network $\mathcal{N} = ExecuteCILP(\mathcal{P})$ that computes $T_{\mathcal{P}}$.*

3.3 Adding Recurrent Links

Next, we extend CILP's architecture for temporal computation making use of delayed recurrent links between output and input layers, as in NARX models [12]. This allows the construction of connectionist models for temporal processing without the need of copying networks in time. To integrate these models we insert a recurrent (delayed) link from any neuron out_α in the network's output (representing an atom α) to a neuron $in_{\bullet\alpha}$ in the network's input. This connection allows the network to propagate the truth value of α at time t to the neuron representing $\bullet\alpha$ at time $t+1$. Fig. 3 presents the algorithm that computes this process.

Lemma 3.4 *Consider a network $\mathcal{N}$ = ExecuteCILP($\mathcal{P}$), where $\mathcal{P}$ is a $\bullet$-based program. The computation of the $\bullet T^t_{\mathcal{P}}$ operator by a network $\mathcal{N}_1$ = InsRecLinks($\mathcal{N}$) is correct if each atom α, such that $\bullet\alpha$ appears in $\mathcal{P}$, is correctly represented by an output neuron out_α in $\mathcal{N}$.*

Proof(sketch): Since the behaviour of the delayed link consists in propagating the output value of a neuron in time t to an input neuron in time $t + 1$, the correctness of the CILP translation algorithm and the semantics of the $\bullet$ operator are sufficient to verify this lemma. □

```
InsRecLinks(N)
    foreach out_α ∈ neurons(N) do
        if in_●α ∈ neurons(N) then
            AddDelayedLink(N, out_α, in_●α)
        end
    end
end
```

Figure 3: Insertion of recurrent links

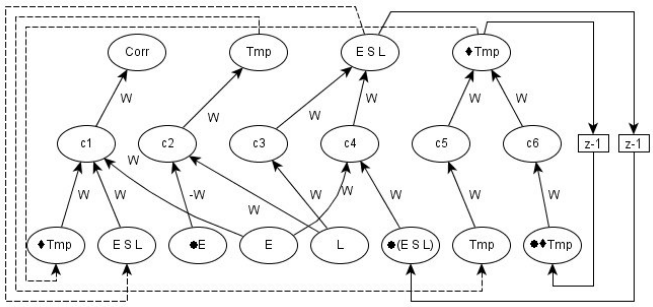

Figure 4: Network generated for the example problem

Fig. 4 shows the architecture produced by the application of the CILP's translation algorithm and the insertion of delayed recurrent links over the program of Table 2. Table 3 shows the fixed point calculated by the network at each time point of **Case a** in Table 1. Values shown in the table consist of the values represented by each output neuron after they converge to a stable state at each time point. This convergence is described in Section 4.

Case a:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
Corr	*F*	*F*	*F*	*F*	*T*	*T*	*T*
Tmp	*F*	*F*	*T*	*T*	*F*	*F*	*F*
E S L	*F*	*F*	*T*	*T*	*T*	*T*	*T*
♦Tmp	*F*	*F*	*T*	*T*	*T*	*T*	*T*

Table 3: Stable states at each time point

3.4 The missing links issue

Due to their behaviour, associated with the $T_{\mathcal{P}}$ operator of a (propositional) logic program, CILP's networks define only the propagation of values to a neuron in the output layer when it represents an atom that appears as head of a clause in the program. Therefore, in order to guarantee the precondition in Lemma 3.4, a mechanism is necessary to ensure that the activation value of an output neuron out_α be correctly associated with the interpretation assigned to the atom α that it represents. This must be achieved even if the activation does not occur by means of the computation of a clause of the program, but through a direct application of a value in an input neuron in_α representing the same atom. Besides the need to represent the propagation of values inserted from temporal links, a similar situation can also be noticed in the original (non-temporal) CILP's model: the value assigned to an atom α, represented by the application of a value in the input neuron in_α representing it has no effect over the output neuron representing the same atom. In the same way, in the model proposed in [11], the output neuron that represents atom α does exist in the network, but it does not correspond to the interpretation of α as there is no connection between this output neuron and the input neuron to which the input value is presented. This is what we call ***the missing links issue***. Table 4 shows the fixed point calculated by the network at each time point of **Case b** in Table 1. Here, one can notice a specific example of the effects of the absence of an output neuron to represent an atom. Since we have no output neuron representing atom E, the input neuron representing $\bullet E$ will not receive any value, therefore the semantics of the atom will not be correctly represented[1]. Our solution to solve this issue consists in inserting clauses in the program, in order to have each necessary atom appearing as head of a clause and keeping the same semantics of the original program, as in Fig. 5. This algorithm uses three different flags for each atom in the program. *IsHead* identifies if the atom is the head of a clause in the program; *IsNeeded* identify if the atom is required in the output; *IsInput* identifies if the atom receives input information (external to the clauses of the program). The later two flags may be defined externally, according the definitions of the problem, or through the algorithm. In our example, *IsNeeded* is *true* for the atom *Corr*, as it represents the output of the system, and *IsInput* is defined as *true* for the two inputs: E and L. After executing the algorithm, one new clause is added to the program: $E \leftarrow E*$.

Case b:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
Corr	*F*	*F*	*T*	*T*	*T*	*T*	*T*
Tmp	*F*	*F*	*T*	*T*	*T*	*F*	*F*
E S L	*F*	*F*	*T*	*T*	*T*	*T*	*T*
♦Tmp	*F*	*F*	*T*	*T*	*T*	*T*	*T*

Table 4: Stable states at each time point

Lemma 3.5 *Consider a $\bullet$-based program $\mathcal{P}_1$, a program $\mathcal{P}_2$ = CorrectProgram($\mathcal{P}_1$) and a network $\mathcal{N}$ = ExecuteCILP($\mathcal{P}_2$). For each atom α that is necessary as output, or that appears in the form $\bullet\alpha$ in program $\mathcal{P}_1$, the network $\mathcal{N}$ contains an output neuron out_α correctly representing α.*

[1] In the example we considered that the absence of value should be represented as a *false* assignment to the atom.

CorrectProgram($\mathcal{P}$)
  foreach $\bullet\alpha \in atoms(\mathcal{P})$ **do**
    $IsNeeded(\alpha) \leftarrow$ **true**
    $IsInput(\bullet\alpha) \leftarrow$ **true**
  end
  foreach $\alpha \in atoms(\mathcal{P})$ **do**
    if $(IsNeeded(\alpha) \vee IsHead(\alpha)) \wedge IsInput(\alpha)$ **then**
      $AddClause(\alpha \leftarrow \alpha*)$
    end
  end
  return $\mathcal{P}$
end

Figure 5: Insertion of clauses in the program

Proof(sketch): There are 3 cases to consider. If the assignment to α is due to a clause in $\mathcal{P}$, then the translation algorithm guarantees the existence of out_α. If the assignment is due to an external assignment, the clause $\alpha \leftarrow \alpha*$ guarantees the existence of out_α and the correct assignment of the external value. If no assignment is defined to α, the atom is not represented on the output layer, and it is assigned *false*. □

3.5 SCTL's Algorithm

The transformation described above for the program requires a slight modification of the last step of the translation algorithm to create correct temporal links to the neuron representing renamed atoms. This algorithm is represented in Fig. 6. Fig. 7 shows the network produced by the algorithm for the

SCTL_Translation($\mathcal{P}$)
  $\mathcal{P}_1 := TempConversion(\mathcal{P})$ $\mathcal{P}_2 := CorrectProgram(\mathcal{P}_1)$
  $\mathcal{N} := ExecuteCILP(\mathcal{P}_2)$ **foreach** $\alpha \in atoms\mathcal{P}_2$ **do**
    if $in_{\bullet\alpha*} \in neurons(\mathcal{N})$ **then**
      $AddDelayedLink(\mathcal{N}, out_\alpha, in_{\bullet\alpha*})$
    end
    else if $in_{\bullet\alpha} \in neurons(\mathcal{N})$ **then**
      $AddDelayedLink(\mathcal{N}, out_\alpha, in_{\bullet\alpha})$
    end
  end
end

Figure 6: SCTL Algorithm

example. A new analysis of the stable states achieved by the network at each time point for cases *a* and *b* in Table 1 shows that the network is computing the expected behaviour of the program, as seen in Table 5.

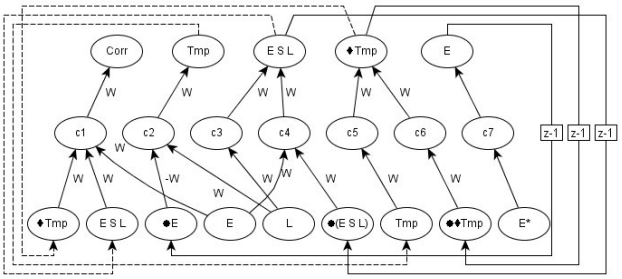

Figure 7: Network generated by the final algorithm

Proposition 3.6 *The network $\mathcal{N}$ generated by SCTL's algorithm over a temporal logic program $\mathcal{P}$ computes the $T^t_\mathcal{P}$ operator of the program at any time point t, provided that the $\mathcal{F}^{t-1}_\mathcal{P}$ operator is correctly calculated.*

The proof of this proposition follows directly from the proofs of lemmas 3.4 and 3.5. Note that one aspect of the translation still must be ensured: the correct computation of the fixed point $\mathcal{F}^t_\mathcal{P}$ operator of the program for each time point *t*. In the next section we show an analysis of the behaviour of the network and propose a method to guarantee this correctness.

Case a:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
$Corr$	F	F	F	F	T	T	T
Tmp	F	F	T	T	F	F	F
$E\ \mathbb{S}\ L$	F	F	T	T	T	T	T
$\blacklozenge Tmp$	F	F	T	T	T	T	T
E	F	F	T	T	T	T	T

Case b:

Atom	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
$Corr$	F	F	F	F	F	F	F
Tmp	F	F	F	F	F	F	F
$E\ \mathbb{S}\ L$	F	F	T	T	T	T	T
$\blacklozenge Tmp$	F	F	F	F	F	F	F
E	F	T	T	T	T	T	T

Table 5: Stable states for modified network

4 Analysing the networks' behaviour

Next, we describe the behaviour of the generated network in order to provide a method to execute the propagation of values and treatment of different recurrent links to guarantee the correct computation w.r.t. semantics. As depicted in the last lines of CILP's algorithm (Fig. 2), some recurrent links are inserted from the output to the input neurons that represent the same atoms. This is done in order to apply, in the input of the network, the values calculated in the feedforward propagation, allowing the recursive calculation of $T_\mathcal{P}$. These recurrent links [11] are used in the CILP networks to achieve a stable state of the network representing the fixed point semantics of the translated program. However, the behaviour of such recurrences and the stability of the model has been hardly studied. When a positive assignment is externally defined for an atom α, it must be kept during all recursive applications of the $T_\mathcal{P}$ operator (until it reaches the fixed point). However, when the positive assignment of the atom is due to a clause of the program, it is a function of a specific calculation of the $T_\mathcal{P}$ operator over a specific input interpretation, and can be changed until the convergence of the system. Therefore, a mechanism to ensure the convergence of the network is needed to allow correct computation of the fixed point of the program by the network, and no practical approaches were defined to do that. To provide such mechanism we focus on acyclic programs. We define the number of executions of the feedforward step of the network that is enough to ensure the computation of the fixed point of the program as follows.

Lemma 4.1 *For each atom α in an acyclic program $\mathcal{P}$, the number $\nu(\alpha)$ of executions of $\mathcal{T}_{\mathcal{P}}(\alpha)$ that is sufficient to converge to a stable interpretation of this atom is a) $\nu(\alpha) = 0$, if α does not appear as head of any clause, or b)$\nu(\alpha) = maxbody(\alpha) + 1$, otherwise. $maxbody(\alpha)$ is defined as the maximum $\nu(\beta)$ among all atoms β that appear in the body of any clause where α is the head.*

Proof(sketch): If an atom α does not appear as head of any clause, its truth value is defined by the external assignment (or assigned to false due to default negation). So, its value is already stable, without the need of any computation of $\mathcal{T}_{\mathcal{P}}$ operator. Else, if α appear as head of one or more clauses, the acyclic limitation to the program ensures that the interpretation of the body of the clauses does not change after "$maxbody(\alpha)$" executions of the $\mathcal{T}_{\mathcal{P}}$ operator. Therefore, only one more execution of the operator is necessary to compute the stable value of α. □

For each acyclic program $\mathcal{P}$, we can define a value $\nu(\mathcal{P})$ that is the greatest value between the $\nu(\alpha)$ of all atoms α in $\mathcal{P}$. In our example, the value of ν for the atoms that appear as head are described in Table 6. Since the value of ν for the remaining atoms is 0, the value of $\nu_{\mathcal{P}}$ for the program is 3, the maximum value in the table.

$\nu(E) = \nu(E*) + 1 = 0 + 1 = 1$
$\nu(Tmp) = max(\nu(L), \nu(\bullet E)) + 1 = 1$
$\nu(\blacklozenge Tmp) = max(\nu(Tmp), \nu(\bullet\blacklozenge Tmp)) + 1 = 2$
$\nu(E \mathbb{S} L) = max(\nu(L), \nu(\bullet(E \mathbb{S} L)), \nu(E)) + 1 = 2$
$\nu(Corr) = max(\nu(E), \nu(E \mathbb{S} L), \nu(\blacklozenge Tmp)) + 1 = 3$

Table 6: Stable states at each time point

Lemma 4.2 *The execution of $\nu_{\mathcal{P}}$ feed-forward steps of a network $\mathcal{N}$ generated by applying CILP's algorithm over an acyclic program $\mathcal{P}$ is equivalent to $\nu_{\mathcal{P}}$ recursive computations of the $\mathcal{T}_{\mathcal{P}}$ operator, and therefore it computes the fixed point of a program $\mathcal{P}$.*

SCTL networks make use of the CILP model to compute the semantics of a logic program, and use delayed recurrent links in order to realise the propagation of past information through time. These links differ from CILP links because the aim of the latter is to allow the recursive computation of the $T_{\mathcal{P}}$ operator at a single time point, i.e. without temporal meaning. The behaviour of SCTL networks can be described as follows. At the beginning of the time flow ($t = 1$), the recurrent links are reset and their value, together with the input vector, are applied to the input layer of the network. During the computation of the same time point, ν feed forward value propagations are executed. Between these propagations, the values in the CILP recurrent links are updated, but the SCTL links are kept unchanged. At the end of the ν propagations [2], a new input vector is presented to the network, and the activation values at the output layer are propagated through SCTL's recurrent links, starting the computation at a new time point. Theorem 4.3 follows from lemma 4.2 and proposition 3.6.

[2]If a training algorithm like backpropagation is applied to the network, the back propagation of the error should be performed in this moment.

Theorem 4.3 *A neural network $\mathcal{N}$ generated by SCTL's translation algorithm application over an acyclic temporal logic program $\mathcal{P}$ computes the fixed point semantics of $\mathcal{P}$.*

5 Conclusions

We have presented a new approach to incorporate past time operators in neural-symbolic systems. We have analysed several aspects concerning the representation of a temporal logic program in a connectionist system and the behaviour of such system when computing the program. This work has also contributed to the *missing links issue* in logic-connectionist translation algorithms. This work can be seen as a stepping stone for the construction of an architecture that integrates past temporal reasoning and learning in neural-symbolic systems. As future work we intend to apply the system to real-life problems so as to verify its adequacy as regards knowledge representation, reasoning and learning in intelligent systems.

Acknowledgements This work has been partly supported by CNPq and CAPES, Brazilian Research Foundations.

References

[1] K. R. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and Computation*, 106:109–157, 1993.

[2] Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard Owens. Metatem: An introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.

[3] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Perspectives in Neural Computing. Springer-Verlag, 2002.

[4] A. S. d'Avila Garcez and Luís C. Lamb. Neural-symbolic systems and the case for non-classical reasoning. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! Essays in honour of Dov Gabbay*, pages 469–488. College Publications, 2005.

[5] A. S. d'Avila Garcez and Luis C. Lamb. A connectionist computational model for epistemic and temporal reasoning. *Neural Computation*, 18(7):1711–1738, 2006.

[6] A. S. d'Avila Garcez, Luis C. Lamb, and D. M. Gabbay. Connectionist computations of intuitionistic reasoning. *Theoretical Computer Science*, 358(1):34–55, 2006.

[7] A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence Journal*, 11(1):59–77, 1999.

[8] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[9] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP*, pages 1070–1080. MIT Press, 1988.

[11] S. Holldobler and Y. Kalinke. Toward a new massively parallel computational model for logic programming. In *Proc. of the Workshop on Combining Symbolic and Connectionist Processing, ECAI 94*, pages 68–77, 1994.

[12] Hava T. Siegelmann, Bill G. Horne, and C. Lee Giles. Computational capabilities of recurrent narx neural networks. Technical report, College Park, MD, USA, 1995.