Learning and Representing Temporal Knowledge in Recurrent Networks

Rafael V. Borges, Artur d'Avila Garcez, and Luis C. Lamb, *Member, IEEE*

***Abstract*— The effective integration of knowledge representation, reasoning, and learning in a robust computational model is one of the key challenges of computer science and artificial intelligence. In particular, temporal knowledge and models have been fundamental in describing the behavior of computational systems. However, knowledge acquisition of correct descriptions of a system's desired behavior is a complex task. In this paper, we present a novel neural-computation model capable of representing and learning temporal knowledge in recurrent networks. The model works in an integrated fashion. It enables the effective representation of temporal knowledge, the adaptation of temporal models given a set of desirable system properties, and effective learning from examples, which in turn can lead to temporal knowledge extraction from the corresponding trained networks. The model is sound from a theoretical standpoint, but it has also been tested on a case study in the area of model verification and adaptation. The results contained in this paper indicate that model verification and learning can be integrated within the neural computation paradigm, contributing to the development of predictive temporal knowledge-based systems and offering interpretable results that allow system researchers and engineers to improve their models and specifications. The model has been implemented and is available as part of a neural-symbolic computational toolkit.**

***Index Terms*— Integrating domain knowledge into nonlinear models, knowledge extraction, model verification, neural-symbolic computation, recurrent neural networks, temporal knowledge learning, temporal logic reasoning.**

I. Introduction

ALTHOUGH nonlinear methods such as neural networks and support vector machines often provide the most accurate predictions, they are generally unsuitable in domains where validation is required because of their black-box nature. This also complicates maintenance and model integration with existing legacy systems. As a result, the use of neural networks has remained restricted in a number of important application areas. White-box models seek to solve this problem in different ways, neural-symbolic computation [1] offers one way of implementing white-box nonlinear prediction. In particular, neural-symbolic systems seek to open the black box by integrating nonlinear modeling with domain knowledge and rule extraction, thus providing insight into the reasoning made by the nonlinear prediction. The construction of such principled integrated models can provide an enriched understanding of the techniques and tools used in neural computation, cognitive science, and artificial intelligence (AI). Specifically, temporal models have been fundamental in these areas. In addition, the problem of knowledge acquisition of sound descriptions of a system's desired behavior is a complex and important task in computer science [2], [3].

In this paper, we present a neural-computation model capable of: 1) representing temporal knowledge operators in recurrent neural networks; 2) adapting temporal knowledge models given a set of desirable system properties; 3) training the networks from examples of system behaviors; and 4) extracting a revised temporal knowledge from the trained networks. In the proposed model, symbolic background knowledge described by a temporal logic formalism is translated into a recurrent neural network. Modified gradient-descent methods are proposed for learning both from examples and system properties, and the trained network can be translated back into a temporal symbolic representation incorporating the initial knowledge revised by the examples and properties. This process is known as the "neural-symbolic cycle" [1], [4], [5].

We have implemented the proposed model as part of a neural-symbolic toolkit and performed experiments on benchmark case studies in the area of model verification and adaptation. The results illustrate how model verification and learning can be integrated within a neural computation paradigm, and indicate that the integration of methodologies from symbolic AI and connectionism is relevant for building robust and sound intelligent systems [1], [3], [6].

Temporal logic has found a large number of applications in computer science [7]–[9]. The importance of adding learning mechanisms to temporal models has been highlighted in several applications, including model discovery and requirements acquisition in software engineering [7], [10], [11]. In what follows, we formally define a correspondence between recurrent networks and temporal logic. We consider the *nonlinear autoregressive exogenous* (NARX) model [12], [13] and define a one-to-one correspondence between NARX and a fragment of temporal logic. We also propose a simple method for the extraction of temporal knowledge from trained NARX networks. As a case study, we consider the problem of software model verification and adaptation, a successful application area of symbolic temporal logic. We have applied our model to the problem of verifying and evolving a specification of a water pumping system [11]. The results indicate that neural-symbolic NARX networks can be used for both verification and learning, reducing the efforts involved in the modeling

Manuscript received May 30, 2010; revised September 16, 2011; accepted September 19, 2011. Date of publication October 17, 2011; date of current version December 13, 2011. This work was supported in part by the Brazilian Research Council CNPq.

R. V. Borges and A. d'Avila Garcez are with the City University London, London EC1V OHB, U.K. (e-mail: Rafael.Borges.1@soi.city.ac.uk; aag@soi.city.ac.uk).

L. C. Lamb is with the Federal University of Rio Grande do Sul, Porto Alegre 91501-970, Brazil (e-mail: LuisLamb@acm.org).

Digital Object Identifier 10.1109/TNN.2011.2170180

process and helping produce verifiable and sound system specifications.

More specifically, we present a translation algorithm that takes temporal knowledge as input (in the form of temporal logic rules) to produce a NARX network. The fragment of temporal logic used is an extension of the logic used in [14] with a richer language containing both future and past operators. Following the neural-symbolic methodology [1], [15]–[19], we then prove that this translation is correct with respect to well-established temporal logic-programming semantics. We then apply a simple pedagogical method [20] for temporal knowledge extraction from trained NARX networks to validate the application. The extraction method is also sufficient for the extraction of trained partial models. This closes the neural-symbolic cycle, allowing the encoding of temporal background knowledge into networks, learning from examples and sequence learning by the networks, and the decoding of the learned models into a revised temporal knowledge for understanding and validation of system properties. The application of the neural-symbolic model to the problems of software model verification and adaptation allows the integration of different dimensions of temporal knowledge, including temporal learning and reasoning about time. The networks are capable of evolving incomplete software specifications from observed examples of system behavior. Furthermore, information about certain desired properties of the system can be verified against the networks by combining the abstract syntax and the verification capacities of a model checking tool with our learning model.

The remainder of this paper is organized as follows. Section II introduces the basics of temporal reasoning, recurrent networks and neural-symbolic computation. Section III presents a language for temporal knowledge representation by recurrent networks, and show the correspondence between the symbolic language and the NARX recurrent networks. Section IV shows how the approach is used for learning from sequences of examples and temporal domain knowledge. In Section V, we apply the approach to a relevant case study showing how the approach can be used for software model verification and adaptation. Finally, we discuss the results, conclude, and point out directions for further research.

II. Background and Related Work

A. Temporal Reasoning

Temporal logics have been highly successful for representing temporal knowledge about computing systems [8]. For example, linear temporal logics (LTL) and computation tree logics (CTL) are broadly used in computer science to analyze models and properties of a system [7], [8]. While LTL uses a linear deterministic approach to the flow of time, CTL allows for the representation of different possible successors for each time point. For simplicity, in this paper we focus on the linear approach, more specifically, we use a specific logic programming language, taking as reference several works that use temporal logics [8], [21]. We shall consider several past and future temporal operators. The past operators include the representation of the previous time point (denoted by $\bullet$), always in the past ($\blacksquare$), sometime in the past ($\blacklozenge$), and the weak and strong variations ($\mathbb{Z}$ and $\mathbb{S}$, respectively) of *since*. Their complementary future operators are, respectively, the next time point operator (denoted by $\bigcirc$), always in the future ($\square$), sometime in the future ($\lozenge$), *unless* ($\mathbb{W}$) and *until* ($\mathbb{U}$), formally defined in the next section.

Model checking is one of the most successful applications of temporal logic. It offers a set of automated tools to perform the formal verification of a system's properties. The system is described as a temporal model so that the satisfiability of a property can be verified automatically. While model checking presents all the advantages of a formal static verification (when compared to the dynamic process of testing), it reduces the need for human intervention [7]. Our experiments include a model checking application as detailed in the sequel. Adding a temporal dimension to the knowledge model imposes some challenges to the task of learning. Symbolic learning systems such as inductive logic programming (ILP) [22] can in principle be adapted for application in temporal domains, but will typically require the use of a correct background knowledge (which may not be possible when dealing, for example, with evolving system specifications). ILP may also turn out to be too brittle for modeling dynamic systems and the task of temporal learning, where a large number of very small adjustments may be required to guarantee robustness, rather than concept-level learning [23].

B. Recurrent Networks

Recurrent networks extend the simple feedforward models by allowing activation propagation to neurons in previous layers, thus adding a loop to the network. As a result, such activation values are considered in future computations of the network. A typical recurrent network used for temporal learning is the *Elman network* [24], which adds neurons in the input layer called *context units* to recurrently receive the output values of hidden neurons. Another way of propagating values through time in neural networks is through *delay units*. Such units output the result of a function applied to the last values received by the input. The most elementary delay unit outputs the value applied to the input at the previous time point. The NARX model has a feedforward core with delay units before the input layer and delayed recurrent links from the output to the input layer. They have been proven equivalent to turing machines [12].

Definition 1: Let $x_i(t)$ denote the value of the ith input neuron at time t. Let $y_j(t)$ denote the value of the jth output neuron at time t. NARX allows the use of $x_i(t)$ and $y_j(t)$ as input at the next time points $t+1$, $t+2$, etc. If x_i is connected to a delay unit z^{-1}, it will be available at $t+1$. A chain of such units can get the value shifted through time and available at $t+2$, etc. It is this variable-size chain that makes NARX convenient for temporal reasoning.

We will use the NARX architecture for both reasoning and gradient-descent learning. Fig. 1 illustrates the model. In the figure, multilayer perceptron (MLP) denotes the feedforward MLP core. In order to train these networks, we use a variation of backpropagation [25] whereby the error is propagated

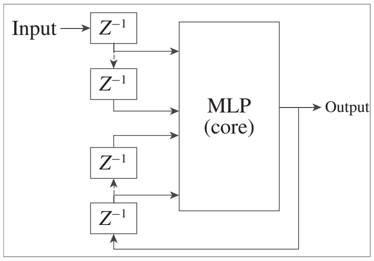


Fig. 1. NARX architecture.

back also through the recurrent connections. In other words, the error component at the input is propagated through the recurrent links to the output neurons in order to be processed by the next backpropagation and weight change step.

C. Neural-Symbolic Computation

Recent studies in AI and evolutionary psychology have produced a number of cognitive models of reasoning, learning, and language that are underpinned by neural computation [26]–[28]. In addition, recent efforts in computer science have led to the development of computational models, called neural-symbolic systems, integrating learning, reasoning, and action [1], [4], [29], [30], including first-order logic systems [31], [32]. Such systems have shown promise in a range of applications, including computational biology, fault diagnosis, fraud prevention [16], and other applications such as, more recently, assessment and training in simulators [33]. The connectionist inductive learning and logic programming (CILP) system [16] is a neural-symbolic system showing a one-to-one correspondence between logic programming and neural networks that are trainable by backpropagation [25].

Definition 2: A logic program is a set of rules of the form $A \leftarrow L_1, L_2, \ldots, L_n$, where A is known as an atom and $L_i (1 \leq i \leq n)$ are called literals. A literal is either an atom (A) or its negation ($\sim A$). A rule like $A \leftarrow L_1, L_2, \ldots, L_n$ states that A is true if L_1 and L_2 and, ..., and L_n are true. When $n = 0$, we have simply $A \leftarrow$, and A is said to be a fact.[1]

The CILP translation from logic programs to neural networks produces single-hidden layer feedforward networks that map each of $L_1, L_2, \ldots, L_n$ to input neurons and A to an output neuron. The networks use a bipolar activation function so that an interval $(-1, -A_{\min}]$ represents truth-value *false*, interval $[A_{\min}, 1)$ represents truth-value *true*, and $(-A_{\min}, A_{\min})$ denotes *unknown*. Positive weights are used to represent positive literals, while negative weights represent negative literals. Hidden neurons implement a logical *and* of the input, and output neurons implement a logical *or* of the hidden neurons. The CILP translation algorithm (described in Fig. 2) sets weights and biases in the network so that the network can be proved equivalent to the original logic program [16]. In other words, the network becomes a computational model for symbolic logic programming. In the algorithm, we have the following parameters.

[1]As is usual, we consider the ground instances of a (first-order) logic program and assume it is finite.

```
CILP_Translation(P)
    foreach C_l ∈ Clauses(P) do
        InsertHiddenNeuron(N, h_l);
        foreach A ∈ Body(C_l) do
            if in_A ∉ Neurons(N) then
                InsertInputNeuron(N, in_A);
                Activation(in_A) ← ψ(x);
            Connect (N, in_A, h_l, W);
        end
        foreach ~ A ∈ Body(C_l) do
            if in_A ∉ Neurons(N) then
                InsertInputNeuron(N, in_A);
                Activation(in_A) ← ψ(x);
            Connect (N, in_A, h_l, −W);
        end
        if out_Head(C_l) ∉ Neurons(N) then
            InsertInputNeuron(N, out_Head(C_l));
        Connect (N, h_l, out_Head(C_l), W);
        Bias (h_l) ← − ((1 + A_min)(k(l) − 1) / 2) W;
        Bias (out_Head(C_l)) ← − ((1 + A_min)(1 − μ_l) / 2) W;
        Activation (h_l) ← φ(x);
        Activation (out_Head(C_l)) ← φ(x);
    end
    foreach A ∈ Atoms(P) do
        if (in_A ∉ Neurons(N)) ∧ (out_A ∈ Atoms(N)) then
            Connect (N, out_A, in_A, 1)
    end
    return N;
end
```

Fig. 2. CILP translation algorithm.

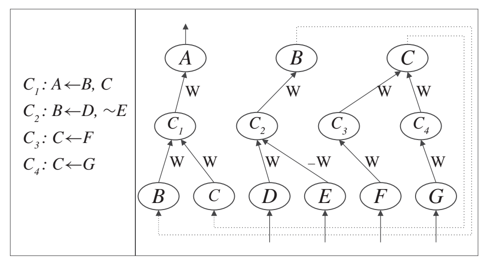


Fig. 3. CILP network.

1) $k(l)$ denotes the number of literals in the body of a clause C_l, $\mu(l)$ is the number of clauses with the same head as C_l.
2) $Max_{k\mu}$ is the maximum among the values of $k(l)$ and $\mu(l)$, and among every clause $C_l \in \mathcal{P}$.
3) $A_{\min}$ is defined in such a way that $(1 - Max_{k\mu}/1 + Max_{k\mu}) < A_{\min} < 1$.
4) $\phi(x)$ is the bipolar sigmoid function $(2/(1 + e^{-\beta x})) - 1$, where β is the parameter that defines the slope of the function, $\psi(x)$ is a linear function (identity).
5) W is the weight of the positive connections, $-W$ is the weight of negative connections. W is defined as a value greater than $(ln(1 + A_{\min}) - ln(1 - A_{\min}))/(Max_{k\mu}(A_{\min} - 1) + A_{\min} + 1) \cdot (2/\beta)$ to guarantee equivalence (see [16] for the proofs).

Fig. 3 shows a CILP network that represents the logic program $A \leftarrow B, C$; $B \leftarrow D, \sim E$; $C \leftarrow F$; $C \leftarrow G$. The CILP system uses the translation to add background knowledge (provided in the form of the logic-program rules) to the neural network. This network can be trained by examples in the usual way. The training examples can change or extend the background knowledge. An extraction algorithm then closes the learning cycle, deriving a revised logic program from the trained network. This process of knowledge revision using neural networks and background knowledge is the main application of the CILP system.

III. Temporal Knowledge in Recurrent Networks

A. Sequential Connectionist Temporal Logic (SCTL)

To allow the representation of temporal knowledge in an integrated reasoning and learning system, we consider the use of a language both simple, to allow the integration with a neural-symbolic engine, and powerful enough to describe sequences of events and the temporal behavior of systems. Thus, we extend the usual logic programming syntax with the modal operators of LTL, as follows.

Definition 3: An expression α is defined as a temporal formula if and only if one of the following holds.

1) $\alpha = A$, where A is a propositional variable.
2) $\alpha = \bullet\beta$, $\alpha = \blacksquare\beta$, $\alpha = \blacklozenge\beta$, $\alpha = \beta\mathbb{S}\gamma$, or $\alpha = \beta\mathbb{Z}\gamma$ (to represent the *past*), where β and γ are also temporal formulas.
3) $\alpha = \bigcirc\beta$, $\alpha = \Box\beta$, $\alpha = \Diamond\beta$, $\alpha = \beta\mathbb{U}\gamma$, or $\alpha = \beta\mathbb{W}\gamma$ (to represent the *future*), where β and γ are temporal formulas.

The operators considered above represent the traditional set of LTL operators, where $\bullet\alpha$ (known as the *yesterday* operator) means that α is true at the previous time point, $\bigcirc\alpha$ (known as the *tomorrow* operator) means that α is true at the next time point, $\blacksquare\alpha$ means that α is always true in the past and $\Diamond\alpha$ means that α will eventually be true in some future point. The $\mathbb{Z}$ and $\mathbb{W}$ binary operators are the weak version of the $\mathbb{S}$ and $\mathbb{U}$ operators, i.e., while $\alpha\mathbb{S}\beta$ represents that α has been true since the last occurrence of β, $\alpha\mathbb{Z}\beta$ will also be true if α has always been true, even if β never occurred. $\mathbb{U}$ (until) and $\mathbb{W}$ (unless) are the future operators corresponding to $\mathbb{S}$ and $\mathbb{Z}$.

Definition 4: A temporal clause is an expression $\alpha_i \leftarrow \lambda_1, \lambda_2, \ldots, \lambda_n$, where α is a temporal formula, and $\lambda_i (1 \leq i \leq n)$ are literals. A literal λ can be either a temporal formula (α) or the negation of a formula ($\sim\alpha$). A temporal logic program $\mathcal{P}$ is a set of temporal clauses.

We will consider that temporal knowledge is defined by a temporal logic program $\mathcal{P}$. In order to define the semantics of the program, we define the operator $\mathcal{T}_\mathcal{P}$ and use the usual fixed-point approach [34]. The semantics of $\mathcal{P}$ is given by an interpretation $\mathcal{F}^t_\mathcal{P}$, which assigns a truth-value to each temporal formula α at each individual time point t. We consider a sequential approach whereby information about the past $\mathcal{F}^{t-1}_\mathcal{P}$ is defined before the current values of $\mathcal{F}^t_\mathcal{P}$ are calculated. By definition, $\mathcal{F}^t_\mathcal{P}$ is a least fixed-point of the meaning operator $\mathcal{T}_\mathcal{P}$ (known as the immediate consequence operator).

B. Formalizing the Temporal Language and Semantics

The $i\mathcal{T}_\mathcal{P}$ operator below defines a consequence relation between the body and the head of the clauses, and the semantics of the $\bullet$ (previous time) and $\bigcirc$ (next time) operators.

Definition 5: The immediate consequence operator $i\mathcal{T}_\mathcal{P}$ of a temporal program $\mathcal{P}$ is a mapping from interpretations to interpretations of $\mathcal{P}$. The application of $i\mathcal{T}_\mathcal{P}$ over an interpretation $I^t_\mathcal{P}$ at a time point t results in a new interpretation at t ($i\mathcal{T}_\mathcal{P}(I^t_\mathcal{P})$) that assigns true to an atom α if any of the conditions below hold.

1) α is head of a clause in the form $\alpha \leftarrow \lambda_1, \lambda_2, \ldots, \lambda_n$, and $I^t_\mathcal{P}(\lambda_1 \wedge \lambda_2 \wedge \ldots \wedge \lambda_n)$ is true.
2) α is an atom in the form $\bullet\beta$, and β is true in $\mathcal{F}^{t-1}_\mathcal{P}$.
3) $\bigcirc\alpha$ is true in $\mathcal{F}^{t-1}_\mathcal{P}$.

In order to derive some properties of this consequence operator, we will need sometimes to restrict $\mathcal{P}$ to programs that admit a single supported model, so that the consequence operator will provably converge to this unique model. Examples of such programs are acyclic programs, as defined below, although the class of such useful programs is more general. The reader is referred to [34] for more details.

Definition 6: The consequence graph $G_\mathcal{P}$ of a program $\mathcal{P}$ is a directed graph defined by a different vertex to represent each different temporal expression α in $\mathcal{P}$. If an expression β (or $\sim\beta$) is in the body of a clause $\alpha \leftarrow \ldots, \beta, \ldots$, then $G_\mathcal{P}$ will contain an edge from the vertex representing β to the vertex representing α (the head of the clause). A program $\mathcal{P}$ is said to be acyclic if $G_\mathcal{P}$ is an acyclic graph.

If $\mathcal{P}$ is acyclic, the recurrent network representing $\mathcal{P}$ will converge in a specific time point t to a fixed point that contains all of the logical consequences of $\mathcal{P}$.

Theorem 7: Given any acyclic temporal program $\mathcal{P}$, $i\mathcal{T}_\mathcal{P}$ converges to a fixed point $i\mathcal{T}^{\nu}_\mathcal{P} = i\mathcal{T}^{\nu-1}_\mathcal{P}$ with $\nu_\mathcal{P}$ given by the maximum length amongst all of the paths in the graph $G_\mathcal{P}$.

Proof: Let G_0 denote the set of vertices in $G_\mathcal{P}$ that are not a target of any edge, i.e., the set of vertices representing expressions not appearing as head of any clause in $\mathcal{P}$. Every expression represented by nodes in G_0 will have a constant value assigned throughout the executions of $i\mathcal{T}_\mathcal{P}$ at t. This value is either given by an input assignment or it is false by default. Let G_1 denote the set of vertices in $G_\mathcal{P}$ that are targets of edges with sources exclusively in G_0. For the expressions represented by the vertices in G_1, a single execution of $i\mathcal{T}_\mathcal{P}$ is sufficient for convergence. This is because the interpretations of the body of these expressions will not change after the first execution. An inductive application of this idea to G_2 (i.e., nodes with edges departing from G_1 and G_0 only), G_3, and so on, is sufficient to prove that the interpretations will converge for every expression, and that the maximum path within $G_\mathcal{P}$ gives the number of executions of $i\mathcal{T}_\mathcal{P}$ that is sufficient to reach such a fixed point. ■

Recall that we use $\mathcal{F}^t_\mathcal{P}$ to denote the fixed point of $i\mathcal{T}_\mathcal{P}$ at each time point t. In order to calculate $i\mathcal{T}_\mathcal{P}$ at a time point t, we assume that $\nu_\mathcal{P}$ executions of $i\mathcal{T}_\mathcal{P}$ were performed at the previous time point $t-1$. We assume a time flow starting at $t = 1$ and a virtual time point $t = 0$ where α is true in $\mathcal{F}^0_\mathcal{P}$ only if α is an expression of the form $\blacksquare\alpha$ or $\alpha\mathbb{Z}\gamma$. Otherwise, α is false in $\mathcal{F}^0_\mathcal{P}$. Let us now define the full set of temporal operators under a consequence operator $\mathcal{T}_\mathcal{P}$. We continue to adopt the sequential approach adopted before.

Definition 8: The immediate consequence operator $\mathcal{T}_\mathcal{P}$ of a temporal program $\mathcal{P}$ is a mapping from interpretations to interpretations of $\mathcal{P}$. The application of $\mathcal{T}_\mathcal{P}$ over an interpretation $I^t_\mathcal{P}$ results in a new interpretation ($\mathcal{T}_\mathcal{P}(I^t_\mathcal{P})$) that assigns true to any atom α when one of the following conditions hold (the definitions below follow the intuitive definitions of the past and future operators as discussed informally earlier, including the variations taking into account the current time point).

1) $i\mathcal{T}_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is true.
2) $\alpha = \blacksquare\beta$ and both $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacksquare\beta)$ and $I^t_{\mathcal{P}}(\beta)$ are true.
3) $\alpha = \blacklozenge\beta$ and either $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacklozenge\beta)$ or $I^t_{\mathcal{P}}(\beta)$ are true.
4) $\alpha = \beta\mathbb{S}\gamma$ and either $I^t_{\mathcal{P}}(\gamma)$ is true or both $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{S}\gamma)$ and $I^t_{\mathcal{P}}(\beta)$ are true.
5) $\alpha = \beta\mathbb{Z}\gamma$ and either $I^t_{\mathcal{P}}(\gamma)$ is true or both $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{Z}\gamma)$ and $I^t_{\mathcal{P}}(\beta)$ are true.
6) $I^t_{\mathcal{P}}(\Box\alpha)$ is true.
7) $\alpha = \Box\beta$ and $\mathcal{F}^{t-1}_{\mathcal{P}}(\Box\beta)$ is true.
8) $\alpha = \Diamond\beta$, $\mathcal{F}^{t-1}_{\mathcal{P}}(\Diamond\beta)$ is true and $\mathcal{F}^{t-1}_{\mathcal{P}}\beta$ is false.
9) There exists some formula β such that $I^t_{\mathcal{P}}(\beta\mathbb{U}\alpha)$ is true and $I^t_{\mathcal{P}}(\beta)$ is false.
10) $\alpha = \beta\mathbb{U}\gamma$, $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{U}\gamma)$ is true and $I^t_{\mathcal{P}}(\gamma)$ is false.
11) There exists some formula β such that $I^t_{\mathcal{P}}(\alpha\mathbb{W}\beta)$ is true and $I^t_{\mathcal{P}}(\beta)$ is false.
12) $\alpha = \beta\mathbb{W}\gamma$, $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{U}\gamma)$ is true and $I^t_{\mathcal{P}}(\gamma)$ is false.

C. Representing SCTL in NARX Networks

To incorporate the above extended semantics into SCTL, we make use of a useful symbolic manipulation. More specifically, we extend the original logic program $\mathcal{P}$ with clauses that can represent the different temporal operators through the use of the $\bullet$ operator. Basically, we use a recursive definition w.r.t. the prior and present time points. In this way, a formula $\blacksquare\alpha$ is true at $t = 1$ if α is true at $t = 0$. $\blacksquare\alpha$ is true at time points $t > 1$ if α is true at t and $\blacksquare\alpha$ is true at $t - 1$. The complete list of definitions is given in the algorithm of Fig. 4. This will allow the representation of any of the temporal operators in the NARX model.

```
Logic_Processing (P)
    foreach α ∈ atoms(P) do
        if α ∈ ■β then
            /* In what follows, AddClause(P,x, y, z) denotes: add
            clauses x, y and z to program P*/
            AddClause(P,□β ← β,●β);
        end
        if α = ◆β then
            AddClause(P, ◆β ← β);
            AddClause(P, ◆β ← ◆●β);
        end
        if α = β𝕊γ then
            AddClause(P,β𝕊γ ← γ);
            AddClause(P,β𝕊γ ← β,●(β𝕊γ));
        end
        if α = βℤγ then
            AddClause(P,βℤγ ← γ);
            AddClause(P,βℤγ ← β,●(βℤγ));
        end
        if α = ○β then
            AddClause(P,β ← ●○β);
        end
        if α =□β then
            AddClause(P,β ←□β);
            AddClause(P,□β ←●□β);
        end
        if α = ◊β then
            AddClause(P,β ← ●○β);
        end
        if α = β𝕌γ then
            AddClause(P,β𝕌γ ← ●(β𝕌γ), ~●(γ));
            AddClause(P,γ ← β𝕌γ,~β);
        end
        if α = β𝕎γ then
            AddClause(P,β𝕎γ ← ●(β𝕎γ), ~●(γ));
            AddClause(P,γ ← β𝕎γ,~β);
        end
    end
end
```

Fig. 4. Logic processing of different temporal operators.

We turn now to showing that the translation obtained from the algorithm of Fig. 4 is logically sound. This result will be needed later to show soundness of the NARX model.

Lemma 9: Let $\mathcal{P}$ and $\mathcal{P}_1$ be temporal logic programs. Let $\mathcal{P}_1$ be the output of the algorithm in Fig. 4 given input $\mathcal{P}$. For every formula α in $\mathcal{P}$, α is true in $\mathcal{T}_{\mathcal{P}}(I^t)$ if and only if α is also true in $i\mathcal{T}_{\mathcal{P}_1}(I^t)$.

Proof: The algorithm adds clauses to the program respecting the semantic definitions of the operators. We can verify this by analyzing each case. Take the case of the $\mathbb{S}$ operator. The first clause inserted ($\beta\mathbb{S}\gamma \leftarrow \gamma$) represents exactly the first option in item 5 of Definition 8. Since $\bullet\alpha$ represents information about α at time point $t - 1$, the clause $\beta\mathbb{S}\gamma \leftarrow \beta, \bullet(\beta\mathbb{S}\gamma)$ represents the second option in the definition of $\mathbb{S}$. The remaining of the proof is as follows: ($\rightarrow$) Assuming that $\mathcal{T}_{\mathcal{P}}(I^t)(\alpha)$ is true, we have two possibilities: if $i\mathcal{T}_{\mathcal{P}}(I^t)(\alpha)$ is true, then clearly the clauses inserted do not change α's truth-value and $i\mathcal{T}_{\mathcal{P}_1}(I^t)(\alpha)$ will be true. If not, a clause will be inserted by the algorithm, and the interpretation of the conjunction of the literals in the body of this clause will be true, thus $i\mathcal{T}_{\mathcal{P}_1}(I^t)(\alpha)$ will be true. ($\leftarrow$) If $\mathcal{T}_{\mathcal{P}}(I^t)(\alpha)$ is false, then none of the clauses inserted by the algorithm will change the interpretation of α, and $i\mathcal{T}_{\mathcal{P}_1}(I^t)(\alpha)$ will also be false. ■

In what follows, we will use the CILP translation to define the feedforward core of the NARX model. We will then make use of the NARX recurrent connections and delay units to implement the temporal operators on top of the feedforward core. As mentioned above, we use a temporal representation based on a sequential approach, where the knowledge about the past is used in the inference of new information about the future. Following this approach, our strategy to represent temporal knowledge is based on the propagation of values through a time flow from a time point $t - 1$ to its subsequent time point t. The semantics adopted for our temporal logic programs follows strictly this idea. Our next step is to see how to implement the delayed propagation of information in the neural-network model.

We have chosen the NARX model because it has: 1) a feedforward core that can be implemented by a CILP translation, and 2) delay units in the input and recurrent connections that can implement the delayed propagation of information needed for temporal reasoning. At each time point t, a new input vector is applied to the network, and all the computations until the network finds a stable state and returns an output are carried out (corresponding to the fixed point of the logic program at that time point). At the next time point $t + 1$, another input vector is applied until the network reaches a stable state producing a new output, and so on. This dynamics of the network is a key difference between SCTL and CILP, where there are no delays or sequence of inputs. In SCTL, the delay units cater for the representation of value propagations over time. For example, this allows a neuron representing a formula $\bullet\alpha$ to receive as input the value of α computed at the previous time point by the network, and produce the correct output. In the same way, an input neuron representing α can receive at t the value of $\bigcirc\alpha$ computed at $t-1$. In this section, we present an algorithm to translate SCTL into NARX and show that the translation is correct w.r.t. the semantics of the temporal operators.

We have considered different ways of representing SCTL in a neural network. The first idea was to use only (delayed) recurrent links to carry the value of an output neuron representing a formula α into an input neuron representing $\bullet\alpha$. When α appears in the head of a clause, the CILP translation generates an output neuron representing α. When α does not appear in the head of a clause, CILP will not have α in the output, and, in the temporal case, it would not be possible to link α to $\bullet\alpha$ and respect the semantics of the $\bullet$ operator. One solution to this is to add clauses of the form $\alpha \leftarrow \alpha$ every time a formula $\bullet\alpha$ appears in a program $\mathcal{P}$ and α is not in the head of any clause in $\mathcal{P}$. In this case, the value of $\nu_{\mathcal{P}}$ is incremented by one due to the insertion of a new clause.

Another approach makes a better use of the available resources of NARX and produces a smaller network. This approach is to use the delay units before the input units to compute the value of $\bullet\alpha$ before computing the value of α in the input. In this case, α does not appear in the output because it is not in the head of any clause. In this way, we avoid having to add clauses to the program and produce a smaller network as a result. For each formula of the form $\bullet^n\alpha$, we insert the delay units as follows (below, we use the notation $operator^n\alpha$ to denote n applications of an operator over α, for example, $\bullet^3\alpha$ denotes $\bullet\bullet\bullet\alpha$).

1) If a formula $\bullet^i\alpha$ appears as head of a clause in $\mathcal{P}$ where $0 \leq i < n$, create a recurrent link from the output neuron representing $\bullet^{\max(i)}\alpha$ to the input neuron representing $\bullet^n\alpha$ and set $n - \max(i)$ as the number of delay units.
2) If no formula $\bullet^i\alpha$ appears as head in $\mathcal{P}$, add n delay units before the input neuron representing $\bullet^n\alpha$ (so that this neuron will receive the value of α at time point $t-n$).
3) If a formula $\bigcirc^n\alpha$ appears as head of a clause in $\mathcal{P}$ ($n > 0$), create a recurrent link from the output neuron representing $\bigcirc^n\alpha$ to the input neuron representing the formula $\bigcirc^i\alpha$ with $\max(i) < n$ and set $n - i$ as the number of delay units.

The algorithm of Fig. 5 takes a temporal logic program as input and produces a NARX model. It is an adaptation of the CILP algorithm and it produces networks with an appropriate set-up of the delay units to implement the temporal constraints.

Theorem 10: Given a temporal logic program $\mathcal{P}$, a NARX neural network $\mathcal{N}$ can be built such that $\mathcal{N}$ computes $\mathcal{T}_{\mathcal{P}}$.

Proof: For the first time point $t = 1$, given arbitrary initial values for the $\bullet\alpha$ formulas, we have that the computation of $\mathcal{T}_{\mathcal{P}}$ is the same as in CILP networks, and it converges to a least fixed point [16]. Inductive step: at a time point t', either $\mathcal{N}$ is stable with α in $\mathcal{F}^{t'}_{\mathcal{P}}(I)$ or the value of α is given as an input. For any formula $\bullet^n\alpha$, if the value of $\bullet^i\alpha$ ($i < n$) is represented in the output of the network, the recurrent link with $n-i$ delay units will apply the correct value to the input of $\bullet^n\alpha$. If $\bullet^i\alpha$ is not represented in the output for any $i < n$, the input value of the neuron $\bullet^n\alpha$ is given by the chain of n delay units in the input. This completes the proof. ■

```
●-based_Translation(𝒫)
    𝒩 ← CILP_Translation(𝒫);
    foreach in_α ∈ Neurons(𝒩) do
        if (α = ●^n β) then
            if ∃i < n (out_{●^i β} ∈ Neurons(𝒩) then
                j ← maximum(i);
                AddDelayLink(𝒩, n − j, out_{●^i β}, in_α);
            else
                AddDelayLink(𝒩, n, in_α);
    end
    return 𝒩;
end
```

Fig. 5. Translation of temporal logic programs into NARX networks.

A corollary of the above theorem is that for acyclic programs, and more generally for any program $\mathcal{P}$ admitting a single supported model, the corresponding recurrent NARX network $\mathcal{N}$ converges to a least fixed point of $\mathcal{T}_{\mathcal{P}}$ denoting the intended meaning of $\mathcal{P}$. We say that $\mathcal{N}$ computes $\mathcal{P}$, in other words, the neural and symbolic representations become interchangeable. In what follows, we exploit this result to allow learning of symbolic temporal knowledge in NARX networks.

IV. Learning in a Temporal Knowledge Domain

Suppose that, in a given application domain, partial symbolic knowledge is available in the form of temporal rules (known as a *model description*). The algorithm of Fig. 5 offers a simple and efficient way of adding knowledge to a NARX model. Suppose, further, that examples are available for training (we call those *observed examples*). In this section, we describe how the NARX model can be trained with such examples. We also consider a third source of information: *system properties*, i.e., properties to be satisfied by the model description.

Let us consider the above three sources of information in the context of an example. In a water pumping system (this is our case study to be discussed in more detail in the next section), an engineer seeks to produce a model description of the system so that it can be implemented and formally verified for errors (since it is a safety-critical system). The engineer starts by defining certain rules, for example, *at any time, if the engine temperature is too high, the pump should be turned off no more than five time steps later*. This is part of the partial model description that can be directly translated into the NARX model by the algorithm of Fig. 5. Producing a sound and complete description is a difficult task, but our engineer might have access to input–output examples that might help (e.g., *at time* t_1 *the temperature was registered as too high, at time* t_2 *the pump was turned off, at time* t_3*, the pump was turned on again, etc*). The sequence of examples may come from the observation of a similar existing system or even from execution logs of the current partial model description itself. These are our observed examples to be trained by backpropagation in the NARX model after the partial description has been translated into it. Finally, the engineer may need to verify certain properties (e.g., *it must* *never be the case where the water level is high and the level of methane is high and yet the pump is on*), and indeed train the NARX model further to try and satisfy these properties when they have not been verified.

In what follows, the temporal logic programs $\mathcal{P}$ will form part of the model description. A model description consists, in addition to the temporal program, of a number of input variables and state variables. Input variables are those whose values are set externally to the model, while state variables have their values defined according to the model's behavior. In the NARX networks, the state variables are represented by the neurons that are recurrently connected. Given observed examples and system properties, sets of input–output examples will be produced for the training of the NARX network. The resulting network is expected to encode a revised model description, be capable of sequence learning and property verification, and produce a final model description that can satisfy the system properties. The learning process will consist of the application of examples to the network in a supervised way. Each training example will be defined as a vector of input values and desired output values in the usual way. An error between desired and obtained network outputs will be minimized through gradient-descent in a backpropagation-like learning process. Below, we give a general definition that includes the case where information about an output is absent.

Definition 11: A model description is a tuple $M = \langle St, In, \mathcal{P}\rangle$, where St is a set of state variables α, In is a set of input variables β, and $\mathcal{P}$ is a set of temporal clauses of the form $\bigcirc\alpha \leftarrow \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$.

Each observed example should assign values to all the input variables and to a subset of the state variables. In other words, the model allows partial observation of state variables. The examples are thus, defined as follows.

Definition 12: An observed example E at time point t is a tuple $E_t = \langle I_t, D_{t+1}\rangle$, where the mapping $I_t : In \rightarrow \{-1, 1\}$ assigns values to the input variables and $D_{t+1} : St \rightarrow \{-1, 0, 1\}$ makes an assignment of desired values to the state variables at the next time point, where 0 denotes that no information is available about the corresponding variable.

As mentioned above, we use gradient descent on the set of tuples $\{E_t\}$, $0 \le t \le n$. First, background knowledge $\mathcal{P}$ can be added to the network using the translation algorithm from the previous section. Then, for observed examples, standard backpropagation applies since each tuple relates t_i with t_{i+1}. For each time point t, the usual two-stage computation takes place. In the forward step, the network computes the next state S_{t+1} given the values of the input vector I_t and the current state S_t (which may be unknown, as defined above). In the backpropagation step, the error is calculated as the difference between S_{t+1} and D_{t+1}, and the weights are adjusted in the usual way [25]. In the case of system properties, the learning above needs to be modified to account for gaps in the sequence of examples, as detailed below. System properties express the expected behavior of a system after an entire sequence of inputs and associated states are presented to the system.

Definition 13: A system property $\mathcal{X}$ is defined by a tuple $\mathcal{X} = \{S_0, I, D_n\}$, where S_0 is a initial state, D_n is a desired final state, and I is a sequence of input variables $I_0, \ldots, I_{n-1}$ with $S_k : St \rightarrow \{-1, 0, 1\}$ and $I_k : In \rightarrow \{-1, 0, 1\}$.

Definition 14: A value assignment to the state variables S_t is said to correspond to a state condition S_k if for every $\alpha \in St$, $S_t(\alpha) = S_k(\alpha)$ or $S_k(\alpha) = 0$. The definition is analogous for input variables.

A property, thus, defines that if the current state of the system at time point t corresponds to S_0, the input applied to the system corresponds to I_0, and thereafter each input applied to the system at time point $t + k$ corresponds to I_k until k is equal to a predefined size n, then the new state of the system S_n must correspond to the desired state D_n. When a value of *zero* is assigned by a state (or input) condition to a variable $\alpha \in S_t$ (or $\beta \in In$), then that condition should not impose any constraint on the value of α (or β).

Property learning requires the propagation of errors through the recurrent connections as described in Section II-B. In the forward step, the network computes state S_1 given the values of the input vector I_0 and the current state S_0, but also S_2 given I_1 and S_1 and so on, up to S_n. In the backpropagation step, the error is calculated as the difference between S_n and D_n and propagated back through the network and its recurrent connections n times before the weights are updated in batch mode in the usual way.

A. System Implementation

We have implemented the above algorithms as part of a unified neural-symbolic system. The system allows the translation of SCTL knowledge into NARX networks, learning from examples and properties, and knowledge extraction from trained NARX networks (discussed in the sequel). Among the several functionalities, it allows the creation of NARX networks from temporal logic programs, as well as the creation of arbitrary architectures of feedforward and NARX networks without background knowledge. The networks can then be subject to learning, with a functionality for evaluating training and test-set performances using cross-validation. The system allows the combination of different sources of information, notably, learning from observed examples and properties. It can also handle learning from multiple properties. Moreover, it includes a tool for automated pedagogical extraction of revised SCTL knowledge from trained NARX networks, rule simplification, and visualization through state-transition diagrams.

When both examples and multiple properties are to be learned simultaneously, the system keeps a record of *active properties* (initially empty) and an index k for each active property. At each time point t, if the current state corresponds to the initial state condition S_0 of a property $\mathcal{X}$, then $\mathcal{X}$ becomes active and k is set to 0. When an input is applied to the network, the system verifies if the input corresponds to the current position In_k of each active property $\mathcal{X}$, eliminating from the list of active properties all the properties not satisfying this condition. When a property becomes inactive, the assignments to the state variables given by the final state condition S_n are used to define the desired output values and then used as part of the learning process. The above mechanism is also used when no examples but only properties are available. In this case, the properties provide the desired output of the network and the inputs to be applied at each time step. At each time point, a property is selected from the list of active properties randomly. When no information is provided about the desired value of a state variable α, the system uses the value obtained by the network as desired value, i.e., $D_{t+1} = S_{t+1}$. This implements a form of expectation maximization. In this way, the error will be null for that neuron and it will not affect the weight correction in the network. Finally, when there is an inconsistency between the values of properties (or between a property and an example), the system adds up the values into a variable sum, and takes $D_{t+1} = 1$ if $sum > 0$, $D_{t+1} = -1$ if $sum < 0$ and $D_{t+1} = 0$ otherwise. Other alternatives are possible here, and might be considered as part of future work. For example, one could assign priorities to properties and rank them in order to mitigate conflicts. The system implementation and the results from our experiments with the water pump case study (described below) are available in http://vega.soi.city.ac.uk/∼abct616/?cont=2.

B. Toward Validating the Model Using Knowledge Extraction

Several approaches to knowledge extraction have been proposed in the literature [5], [35]–[37]. In this paper, extraction is used as a way of validating our model. Below, we sketch the implementation of the extraction tool, which takes a trained NARX network as input and produces temporal logic programs. The implementation is based on pedagogical strategies [20], whereby examples are presented to the network and the obtained outputs are used to define symbolic rules. In pedagogical extraction, one needs to generate a set of examples (input vectors) to be applied to the network. This set must be large enough to offer a good representation of the domain, but not so large that the extraction becomes computationally intractable. Different approaches trying to strike this balance can be found in the literature. In [35], for example, a partial ordering is imposed on the set of input vectors according to the structure of the network so that certain input vectors become preferred over others for querying the network and rule creation. Although not optimized for efficiency, the simple pedagogical approach used here turned out to be sufficient for our purposes of validating the case study, as detailed later.

Consider, first, NARX networks where input information is applied directly to the neurons without delay units and the temporal recurrent links are delayed only by one time point. With these restrictions, at each time point we can associate the input vector I applied to the network to the temporal formulas represented by input neurons. We then run the network once to obtain activation values for the output neurons and, through the recurrent connections, new values for some of the input neurons. Such input neurons that receive information from the output are known as context units. It is useful to distinguish input units (those associated with input vector I) and context units (the values of which define a new state given I). Our system implementation extracts symbolic knowledge from NARX networks by creating a state transition diagram mapping the state of the context units to a new state

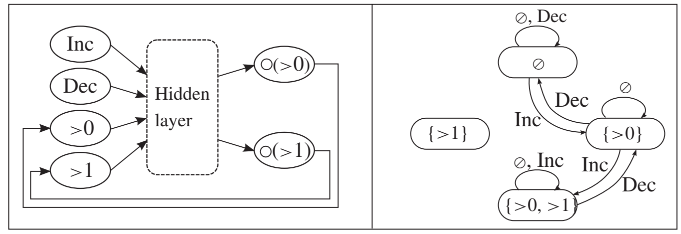


Fig. 6. Example of extraction procedure.

given the input, according to the following definition. Notice that the state diagram is created for visualization purposes, each transition corresponding directly to a temporal rule that can be extracted from the network.

Definition 15: A transition $\mathcal{T}$ is a tuple $\{S_0, I, S_f, w, count\}$ containing a **source** state S_0 and a **target** state S_f given **input** I. Variables w and $count$ are auxiliary information representing a **weight** and the number of occurrences, respectively.

For each time point, a new transition $\mathcal{T}$ is stored: I represents the input vector applied to the network, S_0 contains the values of the context units, and S_f contains the values of the output units. We assign truth-value true (value 1) to positive values in S_f and false (value -1) otherwise, but we use the auxiliary weight w, calculated as a function of the absolute values obtained in the network's output, to calculate a confidence interval on the assignment of truth-values. After a set of inputs is applied to the network, all the occurrences of transition $\mathcal{T}$ with the same S_0, I and S_f are grouped into a single transition $\mathcal{T}'$, where $w^{\mathcal{T}'}$ is the sum of the individual weights and $count^{\mathcal{T}'}$ is the number of transitions grouped. This information is then used to generate a transition diagram that will visually indicate the behavior of the network.

As an example, consider a simple case where an input (Inc) is used to increment the value of a counter, an input (Dec) is used to decrement this value, and the output identifies if the value is greater than zero. Assume that this counter is capable of counting from 0 to 2, and therefore a state variable is needed to record if the value is greater than 1. Fig. 6 shows a network that represents this example on the left hand side, and a state transition diagram extracted from the network on the right hand side. Table I shows a number of extracted state transitions. When grouping the transitions, those in time points $t = 3$ and $t = 8$ will be grouped in a transition $\mathcal{T}$, while the others remain the same (Fig. 6, right-hand side). Besides generating the diagrams, our system implementation also represents the extracted knowledge as temporal logic programs. To do so efficiently, the most important transitions are identified with the use of the auxiliary *weight* and *count* variables. Transitions below a certain number of occurrences or below a desired confidence are removed from the diagram. Each remaining transition $\mathcal{T}'$ is rewritten as a set of clauses—one clause for each output variable. The body of each clause will contain all the input and state variables either in positive or negative form according to the assignments of values to S_0 and I. The head of each clause will be one

TABLE I
TRANSITIONS EXTRACTED FROM THE EXAMPLE IN FIG. 6

	Inputs		State		Outputs		
t	Inc	Dec	> 0	> 1	$\bigcirc(> 0)$	$\bigcirc(> 1)$	$\mathcal{T}$
1	−1	1	−1	−1	−1	−1	$S_0 = \emptyset,\ I = \{Inc\},\ S_f = \emptyset$
2	1	−1	−1	−1	1	−1	$S_0 = \emptyset,\ I = \{Inc\},\ S_f = \{\bigcirc(> 0)\}$
3	−1	−1	1	−1	1	−1	$S_0 = \{> 0\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0)\}$
4	1	−1	1	−1	1	1	$S_0 = \{> 0\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
5	−1	−1	1	1	1	1	$S_0 = \{> 0, > 1\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
6	1	−1	1	1	1	1	$S_0 = \{> 0, > 1\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
7	−1	1	1	1	1	−1	$S_0 = \{> 0, > 1\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0)\}$
8	−1	−1	1	−1	1	−1	$S_0 = \{> 0\},\ I = \{Inc\},\ S_f = \{\bigcirc(> 0)\}$
9	−1	1	1	−1	−1	−1	$S_0 = \{> 0\},\ I = \{Inc\},\ S_f = \emptyset$
10	−1	−1	−1	−1	−1	−1	$S_0 = \emptyset,\ I = \{Inc\},\ S_f = \emptyset$

of the output variables: either $\bigcirc\alpha$, if $S_f(\alpha) = 1$, or $\bigcirc\neg\alpha$, if $S_f^{\mathcal{T}'}(\alpha) = -1$. To allow a better understanding of the rule set, rules obtained from different transitions can also be simplified. A technique based on Karnaugh maps is used, whereby complementary literals can be removed from the body of rules with otherwise the same body and the same head, e.g., $\bigcirc a \leftarrow b, c$ and $\bigcirc a \leftarrow b, \sim c$ can be simplified into a single rule $\bigcirc a \leftarrow b$.

The extraction method can be extended to deal with more delays in the network. For delay units inserted in the input, the rule containing that input neuron will have a $\bullet$ operator for each delay unit in the network. If, for example, literal α is associated with an input neuron with two delay units, $\bullet^2\alpha$ will be used in the rule. The same process can be used for extra delays in recurrent links: if $\bigcirc\bigcirc\alpha$ is associated with output neuron out, and this neuron is connected through two delay units to an input neuron in, then α is added to the rule.

V. CASE STUDY: INTEGRATING SCTL WITH A VERIFICATION TOOL

The intended application of SCTL is in model verification and adaptation. The integration of learning and verification has been considered an important research endeavor [10]. We combine the abstract syntax and the verification capacities of a model checking tool [38] with SCTL/NARX as a representation language and learning system. Model checking tools have three main components: a *description language* used to represent the model; a *specification language* used to represent the properties that should be satisfied by the model; and the *verification engine* that will perform the actual verification. If the model does not satisfy the given properties, the engine will generate a set of counter-examples, i.e., sequences of events where a violation of the property occurs. Below, we integrate all these different sources of information into the SCTL learning system, and show how an iterative process of learning and verification can be used in the revision of temporal models.

In order to illustrate the different steps of the approach, we consider the *pump system testbed* used by [11]. The pump system monitors and controls the levels of water in a mine to avoid the risk of overflow. There are three state variables: $CrMeth$ indicating that the level of methane is critical; $HiWater$ indicating a high level of water; and $PumpOn$ indicating that the pump is turned on. In order to turn on and off such indicators, six different input signals are considered: $sCMOn$ (switch $CrMeth$ on), $sCMOff$ (switch $CrMeth$ off), $sHiW$ (switch $HiWater$ on), $sLoW$ (switch $HiWater$ off), $TurnPOn$ (switch $PumpOn$ on), and $TurnPOff$ (switch $PumpOn$ off). Some of the rules of the system are listed below, where, e.g., if at any time the critical methane switch is turned on then, next time the level of methane indicator will be at critical (first rule). Similarly, if the level of methane indicator is at critical at time t and it is not the case that the pump switch is turned off, then the pump indicator will be on at time $t + 1$ (last rule). Below, $\sim$ stands for (logic programming) negation.

$$\bigcirc CrMeth \leftarrow sCMOn.$$
$$\bigcirc CrMeth \leftarrow CrMeth, \sim sCMOff.$$
$$\bigcirc HiWat \leftarrow sHiW.$$
$$\bigcirc HiWat \leftarrow CrMeth, \sim sLoW.$$
$$\bigcirc PumpOn \leftarrow TurnPOn.$$
$$\bigcirc PumpOn \leftarrow CrMeth, \sim TurnPOff.$$

A. Description Language Used for Verification

Within the logic programming representation, we will consider a fragment of SCTL, allowing the representation of the main aspects of a model checking tool [38]. This fragment satisfies all the properties of the original SCTL language w.r.t. its semantics and translation into NARX. In addition, all the programs in this fragment have the property of being acyclic with $\nu_{\mathcal{P}} = 1$. For simplicity, we restrict the types of variables allowed and assume that the pump system is deterministic (although it should not be too difficult to handle nondeterministic problems given our treatment of unobserved states). An input or state variable can be either Boolean or scalar (i.e., may assume one value from an enumerated set). From now on, it will be useful having a clear distinction between input and state variables. The following slight variation of our temporal logic programs definition captures this formally.

TABLE II
MODEL DESCRIPTION OF THE PUMP SYSTEM

MODULE PumpSystem
IVAR
s : {sCMOn, sCMOff, sHiW, sLoW, TurnPOff};
VAR
CrMeth : **boolean**;
HiWat : **boolean**;
PumpOn : **boolean**;
ASSIGN
init(CrMeth) := **FALSE**;
init(HiWat) := **FALSE**;
init(PumpOn):= **FALSE**;
next(CrMeth) :=
case
s = sCMOn : **TRUE**;
s = sCMOff : **FALSE**;
esac;
next(HiWat) :=
case
s = sHiW : **TRUE**;
s = sLoW : **FALSE**;
esac;
next(PumpOn) :=
case
s = TurnPOn : **TRUE**;
s = TurnPOff : **FALSE**;
esac;

Definition 16: A temporal logic program description $\mathcal{P}$ is a tuple $\mathcal{P} = \{St^{\mathcal{P}}, In^{\mathcal{P}}, Init^{\mathcal{P}}, C^{\mathcal{P}}, Gr^{\mathcal{P}}\}$, where $St^{\mathcal{P}}$ is the set of state variables α, $In^{\mathcal{P}}$ is the set of input variables β, $Init^{\mathcal{P}}$ is the initial state, defined by a mapping from $In^{\mathcal{P}} \cup St^{\mathcal{P}}$ to {true, false} and $C^{\mathcal{P}}$ is a set of clauses in the form $\bigcirc\alpha \leftarrow \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$, denoting that α is true at time t if $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$ is true at time $t-1$. $Gr^{\mathcal{P}}$ is defined as a set of elements in $2^{In^{\mathcal{P}}} \cup 2^{St^{\mathcal{P}}}$.

Our neural-symbolic system implementation contains a module that automatically translates model descriptions provided in the language of a model checker into temporal logic program descriptions. This allows a direct integration of a model checker and SCTL. Given a model description (for completeness we include a description of the pump system in Table II), our goal is to use the model checker to verify system properties and, if a property is violated, use SCTL to revise the description by learning from examples (from Table II, the translation produces temporal rules like the ones above for the pump system).

B. Learning from Counter-Examples and Properties

If all the properties specified are satisfied by the model description, the model checker returns a positive answer and the process can stop. Otherwise, the checker returns what is known as *counter-examples*. These are traces that show why a property has been violated, as formally defined below.

TABLE III
ILLUSTRATION OF COUNTER-EXAMPLES AND THE SEQUENCES FOR LEARNING

t	State	Input
1	$\emptyset$	$sCMon$
2	$\{CrMeth\}$	$sHiW$
3	$\{CrMeth, HiWater\}$	$turnPon$
4	$\{CrMeth, HiWater, PumpOn\}$	

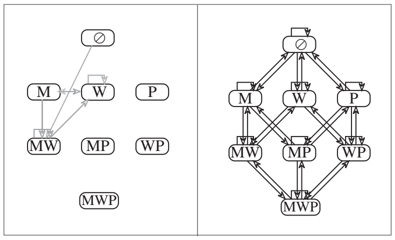
In our case study, these examples will be turned into the training examples used so far to help SCTL learn a new model description. The expectation is that, after a number of iterations, all the properties will eventually be satisfied.

Definition 17: A counter-example $\mathcal{X}$ is defined as a tuple $\mathcal{X} = \{S_0^{\mathcal{X}}, I^{\mathcal{X}}, S_n^{\mathcal{X}}\}$, where $S_0^{\mathcal{X}}$ is the initial state condition, $S_n^{\mathcal{X}}$ is the final state condition, and $I^{\mathcal{X}}$ consists of a sequence of input conditions $(I_0^{\mathcal{X}}, \ldots, I_{n-1}^{\mathcal{X}})$. Each condition assigns a Boolean value to a subset of variables.

A specific state st is said to match a condition $S_i^{\mathcal{X}}$ if, for every variable α with values assigned by $S_i^{\mathcal{X}}$, $S_i^{\mathcal{X}}(\alpha) = st(\alpha)$. The same idea will be used for inputs. Using this idea, counter-examples can produce a large set of sequences to be used for training in SCTL/NARX. Counter-example $\mathcal{X}$ represents that, if the current state of the system at time point t matches $S_0^{\mathcal{X}}$, the applied input matches $I_0^{\mathcal{X}}$, and the following inputs at time point $t+k$ match $I_k^{\mathcal{X}}$ (until k is equal to n), the state of the system must not match $S_n^{\mathcal{X}}$ (notice that each counter-example is a sequence of inputs and states that lead to a violation of a property). In order to train the SCTL/NARX, we negate the final state of $\mathcal{X}$ and use the new sequence ending in $\sim S_n^{\mathcal{X}}$ as a training example in the usual way.

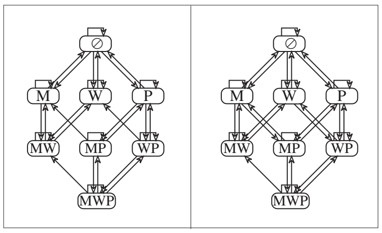
To exemplify this idea, consider the model description of Table II and a safety property expressed in LTL as $G\neg(CrMeth \wedge HiWat \wedge PumpOn)$, meaning that the pump should not be *on* when the level of methane is critical and the water is high. Table III shows the counter-example produced by the checker. From the counter-example, we obtain a new system property $\mathcal{X}'$, such that $S_0^{\mathcal{X}'} = \{\neg CrMeth, \neg HiWat, \neg PumpOn\}$, $I_0^{\mathcal{X}'} = \{sCMon\}$, $I_1^{\mathcal{X}'} = \{sHiW\}$, $I_1^{\mathcal{X}'} = \{turnPOn\}$ and $S_n^{\mathcal{X}'} = \{\neg PumpOn\}$, with $n = 2$. Notice that $\mathcal{X}'$ keeps all the information of the initial state and the sequence of inputs and alters the final state in order to relax the constraint on the variable that regulates the actual state of the pump, in this case. Alternative more sophisticated methods of generating positive examples from counter-examples exist [39], and may be considered as part of future work. The SCTL/NARX learning process could be greatly facilitated, if, for example, the intervention of an expert was possible at this stage. An expert could identify undesirable states in the middle of a counter-example sequence and propose better positive examples than the above, or reduce the specificity of the counter-example to the right level in one fell swoop by identifying a number of undesired cases in one goal.

Let us now use the pump system to illustrate the complete iterative process of verification and learning. A sequence of



(a) (b)

Fig. 7. Effects of learning from examples in the transition diagrams. (a) Initial model extracted from network before training. (b) Diagram extracted after training.



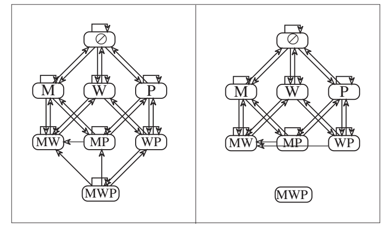
(a) (b)

Fig. 8. Effects of learning from counter examples (a) and (b) show two possible outcomes of training a network representing the knowledge from Fig. 7(a) with the counter example of Table III.

1000 input–output patterns were used in our experiments. All the state variables were observable and the examples were generated from the model description in Table II. A NARX network was created without any background knowledge and was subject to the successive presentation of these examples. Fig. 7 shows a state transition diagram representing the knowledge extracted from the network before [Fig. 7(a)] and after [Fig. 7(b)] the network was trained. In Fig. 7, M represents *critical methane* ($CrMeth$), W represents *high water* ($HiWat$), and P represents that the pump is on ($PumpOn$). As can be observed in Fig. 7, the NARX starts with a set of random transitions with low weight, represented in lighter shades. As learning progresses, it adapts to represent stronger and more robust transitions. If we convert this transition diagram into a temporal logic program representation, we obtain a similar description as the original one provided above, indicating that the network manages to learn the rules from examples. Notice that, if the rules are available, they can be translated directly into the network, without the need for training those 1000 examples.

Next, let us add to the training the new system property obtained from the counter-example of Table III. In this part of the experiment, we compare the network trained from the examples and the new property with a network created by translating the original rules above and then trained with the new property. Fig. 8 shows the transition diagrams extracted in either case. Notice that in diagram a, the only situation where the pump switches from off to *on* is when both $CrMeth$ and $HiWat$ are false. In diagram b, the only change is in the case where both variables $CrMeth$ and $HiWat$ are true.

Considering case b to continue our analysis, one can represent the trained network (with extracted rules) in the form of



(a) (b)

Fig. 9. Transition diagrams representing effects of iterating properties. (a) Result of learning the counter example of Table IV over the model of Fig 8(b). (b) Result of learning the counter example of Table V over the model of (a).

TABLE IV

NEW COUNTER-EXAMPLE

t	State	Input
1	$\sim CrMeth, \sim HighWater \sim PumpOn$	$sCMon$
2	$\{CrMeth, \sim HighWater, \sim PumpOn\}$	$turnPon$
3	$\{CrMeth, \sim HighWater, PumpOn\}$	$sHiW$
4	$\{CrMeth, HighWater, PumpOn\}$	–

TABLE V

FINAL COUNTER-EXAMPLE

t	State	Input
1	$\sim CrMeth, \sim HighWater \sim PumpOn$	$sHiW$
2	$\{\sim CrMeth, HighWater, \sim PumpOn\}$	$turnPon$
3	$\{\sim CrMeth, HighWater, PumpOn\}$	$sCrMeth$
4	$\{CrMeth, HighWater, PumpOn\}$	–

a new model description. As can be seen from the figure, the new description includes a new condition when turning the pump on. This learned condition does not include the input telling the pump to turn on when the water is high and the methane is at a critical level. It is therefore general enough to deal with different sequences than the one provided in the counter-example. However, the system still does not deal with the case where the pump needs to be turned off because a new input leads to an undesired state. In other words, the new model description still does not satisfy the safety property; this can be verified by a second running of the model checker, as described below.

C. Iterating Verification and Learning

Early work on the integration of verification and learning indicates that a cycle of analysis and revision might converge to a correct specification that satisfies system properties [40]. Our proposal in this paper follows this idea. Therefore, we apply the model checking tool to verify the same property, now on the revised model description. A new counter-example is obtained (Table IV).

From the new counter-example, we define a new sequence for training: $\{\} \rightarrow sCMOn \rightarrow sHiW \rightarrow TurnPOn \rightarrow \{\sim PumpOn\}$. After this, the diagram shown in Fig. 9(a) was extracted. One can see that the original LTL property is still not satisfied. After verification again, this time we obtain a final counter-example (Table V). After adapting to this final counter-example, we finally obtain the diagram shown in Fig. 9(b). When applying the model checker to this new description, the property is finally satisfied (as should be already clear from the diagram).

VI. Conclusion

We have presented a novel neural-computational model capable of representing and learning temporal knowledge in different domains. The white-box methodology presented here is based on solid ideas from AI, cognitive science, and neural computation. The use of a neural-symbolic approach enables the integration of temporal domain knowledge into a non-linear recurrent network model, learning from sequences, counter-examples and system properties, and temporal logic rule extraction from the trained models. The extracted rules can also be visualized through the use of a state diagram tool, and a cycle of learning and verification was implemented through the translation of the model checker into the model. The use of the neural-symbolic methodology enables the use of recurrent networks in domains where traditionally only symbolic methods were used. We seek to promote a robust and effective learning of temporal representations through the use of a connectionist model of computation, yet maintaining sound temporal reasoning and transparency as required by the application. The main results presented in this paper are as follows.

1) A formal approach that allows the integration of temporal knowledge representation, learning and reasoning into a unified model, making use of a robust connectionist approach for learning, but also providing tools to integrate background information and extracting the learned knowledge. Therefore the proposed methodology overcomes some of the strongest criticisms to neural networks found in literature.
2) The use of rich testbeds as a clear demonstration of the different steps involved in the proposed framework. The results obtained with the learning (and extraction) steps are empirical evidence of the learning capabilities of SCTL. In particular, the results corroborate the importance of adding background knowledge (when available) into neural networks learning.
3) The novel application of our methodology in the verification and revision of software models, providing an automated tool for the different processes as highlighted by [11] and [40]–[42], in addition, the integration with an existing model checking tool with several functionalities has led to results clearly useful in relevant application domains [38].

Limitations of the approach include, as discussed and analyzed throughout this paper, the difficulty in fully automating the entire process, in particular the process of converting counter-examples into useful training sequences for learning. Extraction is generally perceived as the bottleneck of the neural-symbolic methodology and this is no exception in this paper. Perhaps, it is even more so in the case of recurrent networks. Nevertheless, the extraction and validation of partial models has been possible. This opens up a number of research avenues in the area of rule extraction from recurrent networks, which may lead to a range of new applications, as suggested in [39]. In summary, we believe that this paper has described a rich methodology for temporal knowledge representation, learning, and verification, shedding new light on predictive temporal models not only from a theoretical standpoint, but also with respect to a potentially large number of applications in computational intelligence, software engineering, neural computation, and cognitive science.

Acknowledgment

The authors would like to thank the anonymous referees and P. Hitzler for their suggestions which helped improve the presentation of this paper.

References

[1] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay, *Neural-Symbolic Cognitive Reasoning* (Cognitive Technologies). New York: Springer-Verlag, 2009.
[2] E. A. Feigenbaum, "Some challenges and grand challenges for computational intelligence," *J. ACM*, vol. 50, no. 1, pp. 32–40, Jan. 2003.
[3] L. G. Valiant, "Three problems in computer science," *J. ACM*, vol. 50, no. 1, pp. 96–99, Jan. 2003.
[4] S. Bader and P. Hitzler, "Dimensions of neural-symbolic integration–a structured survey," in *We Will Show Them! Essays in Honour of Dov Gabbay* (International Federation for Computational Logic), S. Artëmov, H. Barringer, A. d'Avila Garcez, L. Lamb, and J. Woods, Eds. Tucson, AZ: College Publications, 2005, pp. 167–194.
[5] J. Lehmann, S. Bader, and P. Hitzler, "Extracting reduced logic programs from artificial neural networks," *Appl. Intell.*, vol. 32, no. 3, pp. 249–266, 2010.
[6] I. Cloete and J. Zurada, *Knowledge-Based Neurocomputing*. Cambridge, MA: MIT Press, 2000.
[7] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *Commun. ACM*, vol. 52, no. 11, pp. 74–84, Nov. 2009.
[8] M. Fisher, D. Gabbay, and L. Vila, *Handbook of Temporal Reasoning in Artificial Intelligence*. Amsterdam, The Netherlands: Elsevier, 2005.
[9] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symp. Found. Comp. Sci.*, Providence, RI, Oct.–Nov. 1977, pp. 46–67.
[10] D. Zhang and J. Tsai, *Machine Learning Applications in Software Engineering*. River Edge, NJ: World Scientific, 2005.
[11] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *Proc. ICSE*, May 2009, pp. 265–275.
[12] H. T. Siegelmann, B. G. Horne, and C. L. Giles, "Computational capabilities of recurrent NARX neural networks," *IEEE Trans. Syst., Man Cybern., Part B*, vol. 27, no. 2, pp. 208–215, Apr. 1997.
[13] T. Lin, B. Horne, P. Tino, and C. L. Giles, "Learning long-term dependencies in NARX recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 7, no. 6, pp. 1329–1338, Nov. 1996.
[14] L. C. Lamb, R. V. Borges, and A. S. d'Avila Garcez, "A connectionist cognitive model for temporal synchronization and learning," in *Proc. 22nd AAAI Conf. Artif. Intell.*, 2007, pp. 827–832.
[15] R. Sun, "Robust reasoning: Integrating rule-based and similarity-based reasoning," *Artif. Intell.*, vol. 75, no. 2, pp. 241–295, Jun. 1995.
[16] A. d'Avila Garcez, K. Broda, and D. Gabbay, *Neural-Symbolic Learning Systems: Foundations and Applications* (Perspectives in Neural Computing). New York: Springer-Verlag, 2002.
[17] A. d'Avila Garcez and L. Lamb, "A connectionist computational model for epistemic and temporal reasoning," *Neural Comput.*, vol. 18, no. 7, pp. 1711–1738, Jul. 2006.
[18] A. d'Avila Garcez, L. Lamb, and D. Gabbay, "Connectionist computations of intuitionistic reasoning," *Theoret. Comput. Sci.*, vol. 358, no. 1, pp. 34–55, Jul. 2006.
[19] A. d'Avila Garcez, M. D. Gabbay, and S. Artur, "Connectionist modal logic: Representing modalities in neural networks," *Theoret. Comput. Sci.*, vol. 371, nos. 1–2, pp. 34–53, 2007.

[20] R. Andrews, J. Diederich, and A. B. Tickle, "A survey and critique of techniques for extracting rules from trained artificial neural networks," *Knowl.-Based Syst.*, vol. 8, no. 6, pp. 373–389, Dec. 1995.

[21] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens, "METATEM: An introduction," *Formal Asp. Comput.*, vol. 7, no. 5, pp. 533–549, 1995.

[22] S. Muggleton and L. Raedt, "Inductive logic programming: Theory and methods," *J. Logic Program.*, vol. 19, no. 20, pp. 629–679, 1994.

[23] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[24] J. L. Elman, "Finding structure in time," *Cognit. Sci.*, vol. 14, no. 2, pp. 179–211, 1990.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Parallel Distrib. Process.*, vol. 1, pp. 318–362, Oct. 1986.

[26] S. Pinker, M. A. Nowak, and J. J. Lee, "The logic of indirect speech," *Proc. Nat. Acad. Sci.*, vol. 105, no. 3, pp. 833–838, Jan. 2008.

[27] L. Shastri, "Shruti: A neurally motivated architecture for rapid, scalable inference," in *Perspectives of Neural-Symbolic Integration*, B. Hammer and P. Hitzler, Eds. New York: Springer-Verlag, 2007, pp. 183–203.

[28] R. Sun, "Theoretical status of computational cognitive modeling," *Cognit. Syst. Res.*, vol. 10, no. 2, pp. 124–140, 2009.

[29] L. Valiant, "A neuroidal architecture for cognitive computation," *J. ACM*, vol. 47, no. 5, pp. 854–882, 2000.

[30] B. Hammer and P. Hitzler, *Perspectives of Neural-Symbolic Integration*. New York: Springer-Verlag, 2007.

[31] H. Gust and K.-U. Kuehnberger, "Learning symbolic inferences with neural networks," in *Proc. 27th Ann. Conf. Cognit. Sci. Soc.*, 2005, pp. 875–880.

[32] S. Bader, P. Hitzler, S. Holldobler, and A. Witzel, "A fully connectionist model generator for covered first-order logic programs," in *Proc. Int. Joint Conf. Artif. Intell.*, 2007, pp. 666–671.

[33] L. de Penning, A. S. d'Avila Garcez, L. C. Lamb, and J. J. Meyer, "A neural-symbolic cognitive agent for online learning and reasoning," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, Jul. 2011, pp. 1653–1658.

[34] P. Hitzler and A. K. Seda, "Characterizations of classes of programs by three-valued operators," in *Proc. 5th Int. Conf. Logic Prog. Non-Monotonic Reason.*, 1999, pp. 357–371.

[35] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay, "Symbolic knowledge extraction from trained neural networks: A sound approach," *Artif. Intell.*, vol. 125, nos. 1–2, pp. 155–207, 2001.

[36] R. Setiono, W. Leow, and J. Zurada, "Extraction of rules from artificial neural networks for nonlinear regression," *IEEE Trans. Neural Netw.*, vol. 13, no. 3, pp. 564–577, May 2002.

[37] R. Setiono, B. Baesens, and C. Mues, "Recursive neural network rule extraction for data with mixed attributes," *IEEE Trans. Neural Netw.*, vol. 19, no. 2, pp. 299–307, Feb. 2008.

[38] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model verifier," in *Proc. Int. Conf. Comput. Aided Verificat.*, 1999, pp. 495–499.

[39] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens, "Robust process discovery with artificial negative events," *J. Mach. Learn. Res.*, vol. 10, pp. 1305–1340, Jun. 2009.

[40] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer, "An analysis-revision cycle to evolve requirements specifications," in *Proc. 16th IEEE Int. Conf. Autom. Softw. Eng.*, San Diego, CA, Nov. 2001, pp. 354–358.

[41] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel, "Using abduction and induction for operational requirements elaboration," *J. Appl. Logic*, vol. 7, no. 3, pp. 275–288, 2009.

[42] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *J. Autom. Lang. Combinator.*, vol. 7, no. 2, pp. 225–246, 2001.

Rafael V. Borges received the B.Sc. degree from the Federal University of Pelotas, Pelotas, Brazil, and the M.Sc. degree from the Federal University of Rio Grande do Sul, Porto Alegre, Brazil, in 2004 and 2007, respectively, both in computer science. He is currently pursuing the Ph.D. degree with the City University London, London, U.K.

His current research interests include integration of the symbolic and connectionist paradigms of artificial intelligence and formal methods of model specification and verification in software engineering.

Artur S. d'Avila Garcez received the Ph.D. degree in computer science from Imperial College London, London, U.K., in 2000.

He is a Reader in neural-symbolic computation with the School of Informatics, City University London, London. He has co-authored two books: *Neural-Symbolic Cognitive Reasoning* (Springer, 2009) and *Neural-Symbolic Learning Systems: Foundations and Applications* (Springer, 2002). His research papers have appeared in the *Behavioral and Brain Sciences*, *Theoretical Computer Science*, *Neural Computation*, *Journal of Logic and Computation*, *Artificial Intelligence*, and *Studia Logica*. His current research interests include neural computing, artificial intelligence, and computer science logic.

Dr. Garcez is a fellow of the British Computer Society.

Luis C. Lamb (M'07) received the Ph.D. and Diploma degrees in computer science from Imperial College London, London, U.K., in 2000.

He is an Associate Professor and Deputy Dean of the Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, Brazil. He is an Honorary Visiting Fellow with the Department of Computing, City University London, London. He has co-authored two research monographs: *Neural-Symbolic Cognitive Reasoning* (Springer, 2009) and *Compiled Labelled Deductive Systems* (IoP, 2004). His research has led to publications in *Theoretical Computer Science, Neural Computation*, the *Philosophical Transactions of the Royal Society A, ACM Transactions on Autonomous and Adaptive Systems, Behavioral and Brain Sciences, Physica A*, and the *Journal of Theoretical Biology*. His current research interests include logic in computer science, artificial intelligence, computing in the physical and social sciences, and neural and cognitive computation.

Dr. Lamb holds an Advanced Research Fellowship from the Brazilian Research Council CNPq. He is a professional member of the Association for the Advancement of Artificial Intelligence, the Association for Computing Machinery, the American Mathematical Society, the Association for Symbolic Logic, and the Brazilian Computing Society.