Neural-Symbolic Learning and Reasoning
AAAI Technical Report WS-12-11

Neural-Symbolic Rule-Based Monitoring

Alan Perotti
University of Turin
perotti@di.unito.it

Artur d'Avila Garcez
City University London
aag@soi.city.ac.uk

Guido Boella
University of Turin
boella@di.unito.it

Daniele Rispoli
University of Turin
daniele.rispoli@gmail.com

Abstract

In this paper we present a neural-symbolic system for monitoring traces of observations in sofware systems. To this end, we define an algorithm that translates a RuleR rule-based monitoring system (RS) into a rule-based neural network system (RNNS). We then show how the RNNS can perform trace monitoring effectively and analyze its performance, reporting promising preliminary results. Finally, we discuss how network learning could be used within RNNS to embed the system into a framework for iterative verification and model adaptation. It is hoped that a tight integration of verification and adaptation within the neural-symbolic approach will help support the development of self-adapting, self-healing systems.

1 Introduction

Several results in the area of neural-symbolic integration have shown how the embedding of symbolic knowledge into a connectionist model (such as a neural network) can provide a flexible, parallelizable environment for effective learning and reasoning (see (d'Avila Garcez, Broda, and Gabbay 2002) for an introduction).

In this paper, we argue that adopting the neural-symbolic methodology can be advantageous also in the area of runtime monitoring and adaptation of software systems (Barringer, Rydeheard, and Havelund 2010). Rule systems allow the verification of system properties by checking whether sequences (traces) of activities comply with the properties, one trace step at a time. We show that neural-symbolic integration can offer an iterative framework for fast monitoring of properties and model revision. A recent and powerful rule-based monitoring tool is RuleR (Barringer, Rydeheard, and Havelund 2010). In this paper, we focus on the creation of the neural-symbolic monitoring system from RuleR, and leave the integration of model revision as future work. However, given the nature of the neural-symbolic approach, this integration should be straightforward (although experimental evaluations will obviously be needed). A direct link between RuleR and neural networks is established by a translation algorithm first introduced in this paper.

In a nutshell, we address the following research question: *how can we perform rule-based trace monitoring within a neural-symbolic system?* We tackle this question by (i) defining a translation from rule-based monitoring system RuleR into a connectionist network model and (ii) showing how one can use the resulting network model to perform trace monitoring efficiently; we analyze the performance of an implementation of our system on a benchmark example, checking accuracy and measuring system scalability.

The paper is organized as follows: in Section 2 we describe the relevant background about the neural-symbolic approach and the RuleR monitoring system. Section 3 introduces the encoding of a rule system in RuleR into a neural network, and Section 4 defines how the network can be used for online trace monitoring. Section 5 gives an example of translation and monitoring, and Section 6 discusses the experiments we did to test our implementation. Section 7 discusses directions for future work, and Section 8 concludes the paper.

2 Background

This paper presents an integration of the monitoring system RuleR (Barringer, Rydeheard, and Havelund 2010) with the paradigm of neural-symbolic computation (d'Avila Garcez, Broda, and Gabbay 2002). In this section, we briefly summarize the two.

The main purpose of a neural-symbolic system is to bring together the connectionist and symbolic approaches exploiting the strengths of both paradigms and, hopefully, avoiding their drawbacks. In (Towell and Shavlik 1994), Towell and Shavlik presented the influential neural-symbolic system KBANN (Knowledge-Based Artificial Neural Network), a system for rule insertion, refinement and extraction from feedforward neural networks. KBANN served as inspiration for the construction of the Connectionist Inductive Learning and Logic Programming (CILP) system (d'Avila Garcez, Broda, and Gabbay 2002; d'Avila Garcez and Zaverucha 1999). CILP builds upon KBANN and (Hoelldobler and Kalinke 1994) to provide a sound theoretical foundation for reasoning in artificial neural networks and learning capabilities; the general framework of a neural-symbolic system is composed by three main phases: encoding symbolic knowledge in a neural network, performing theory revision (by means of some learning algorithm) in the network, and extracting a revised knowledge from the trained network. In what follows, we use a variant of CILP since we are interested in the automated integration of verification (through reasoning, model checking) and adaptation (through learning) of software systems and models. More recently, Borges et al. (Borges et al. 2011; Borges, d'Avila Garcez, and Lamb 2011) proposed a new neural-symbolic system, named Sequential Connectionist Temporal Logic (SCTL) for integrating verification and adaptation of software descriptions. This framework encodes a system description into a particular kind of network, namely NARX (Nonlinear AutoRegressive with eXogenous inputs) and then a learning phase allows the integration of different knowledge sources. Properties to be satisfied are checked by means of an external model checking tool (such as NuSMV): if a property is not satisfied, the tool provides a counterexample that can be adapted with the help of an expert and be used to train the NARX network to hopefully adapt the model in the right direction. In this paper, we argue that the simpler networks used by CILP can achieve similar results, and show that a variation of CILP should be able to implement the entire process without the need for using an external model checking tool.

RuleR is a rule-based monitoring system. A wide range of temporal logics and other specification formalisms can be compiled into RuleR rules, and the obtained rule system can be used for run-time verification. RuleR relies on the *declarative past and imperative future* paradigm in executable logic (Gabbay 1987); in its simpler version, RuleR encodes propositional LTL, with logical formulae normalized in Separated Normal Form (Fisher 1997). RuleR is defined as a tuple $\langle R, O, P, I, F\rangle$. R are rule names and O observation names. Rules (defined in P) are formed from a condition and a body, the fist one being a conjunction of literals and the second a disjunctive set of literals; literals are (potentially negative) occurrences of rule names or observation names. The main idea is that rules can activate/deactivate other rules at any iteration. When a rule is active, if its condition evaluates to true for the current state (formed from the current observations and previous obligations of the rule system), its body defines what rules are activated in the next state and what observations must hold. Together with rules and observations, a monitoring system is composed of initial rules (I) and forbidden ones (F). The monitoring of a trace follows an iterative algorithm: given a set of initial states, at each iteration the set of new observations (representing a step in the trace) is given. Inconsistent states are removed, active rules are triggered and a new frontier is obtained. At the end of the trace, all states containing forbidden rules are removed, and if at least one state remains then the trace is said to satisfy the rule set. As an example, consider the rule $r_0 : \longrightarrow a, r_0 \mid b$.

Suppose that r_0 is in the initial and in the forbidden set. This rule system is formally defined as

$$\langle\{r_0\}, \{a, b\}, \{r_0 : \longrightarrow a, r_0 \mid b\}, \{\{r_0\}\}, \{r_0\}\rangle$$

and it models the LTL formula $a\ until\ b$. The system starts with r_0 active, and that rule can either reactivate itself together with the future-time obligation a or just impose the obligation b. Note that if neither a nor b are observed, no consistent state is created and the monitoring fails. If b is observed, r_0 is not activated anymore: since r_0 is the (only) forbidden rule, the trace satisfies the system iff r_0 is not activated at the end of the trace, that is, if b has been observed at least once after a sequence of a's.

3 From Rule Systems to Neural Networks

In this section, we describe how we encode RuleR rules into feedforward, single-hidden layer networks. In the spirit of CILP, these networks will then be turned into recurrent networks to achieve the parallel computation of the rules. RuleR rules (from now on simply called "rules" for convenience) are composed by a *name*, a *condition* and a *body*:

$$name\ :\ condition \longrightarrow body$$

They are often represented as tuples:

$$\langle\{name\}, \{condition\}, \{body\}\rangle$$

Given a rule $\langle r, C, B\rangle$, we may refer to its condition as $C(r)$ and to its body as $B(r)$, respectively. Conditions are conjunctions of literals, while bodies are disjunctions of conjunctions of literals; literals are (potentially negated) occurrences of rule names and observation names. A rule system is a tuple $\langle R, O, P, I, F\rangle$ such that R is a set of rule names, O a set of observation names, P is the actual description of the rules, I is a set of possible initial states and F is the set of forbidden rules.

3.1 Network architecture

Algorithm 1 below describes the steps for encoding a rule system into the neural network. Note that all neurons are labelled with the (potentially negated) name of the literals in the rule system. We use x_i to indicate the neuron with label x in the *input* layer (similarly for the hidden and output layers); given a set of literals L, $L^{\pm}$ is its negative closure. As in CILP, the main idea is to map conditions to the input layer, rules to hidden neurons and bodies to the output layer.

Algorithm 1 Translation algorithm

```
 1: for all $x \in \{R^{\pm} \cup O^{\pm}\}$ do
 2:     add a neuron (labelled $x$) in the input layer.
 3:     add a neuron (labelled $x$) in the output layer.
 4:     add a recurrent connection from $x_o$ to $x_i$
 5: end for
 6: for all $x \in R$ do
 7:     add a neuron (labelled $x$) in the hidden layer.
 8:     add a recurrent connection from $x_o$ to $x_h$
 9: end for
10: for all $\langle r, C, B\rangle \in P$ do
11:     for all $c \in C$ do
12:         add a connection from $c_i$ to $r_h$.
13:     end for
14:     for all $b \in B$ do
15:         add a connection from $r_h$ to $b_o$.
16:     end for
17: end for
```

The behaviour of a rule (that is, a satisfied condition triggering the body) is mirrored in a signal propagation from the input layer to the ouput layer through the activation of the hidden neuron representing the rule.
Lines 1,2,3 of the algorithm create, both in the input and output layer, a neuron for every rule or observation name and its negation. We represent negation explicitly since the monitoring process requires the network to detect (and remove) inconsistencies (hence, modelling negation with the lack of signal would not allow us to do this). Lines 6,7 create a hidden neuron for each rule name. Lines 1,4 add a normal recurrent connection from the output to the input layer; lines 6,8 add a recurrent connection from the output to the hidden layer (differently from CILP). This connection is used to modify the hidden neuron's threshold and it models the rule activation: its use will be explained in what follows. Lines 10 to 16 create the feedforward connections from the condition literals to the rule names and from the names to the body literals.

3.2 Rule-based Neural Network System

We now define the network system more formally.

Definition 1. *A* **Rule-based Neural Network** RNN *is a network, obtained from a RuleR rule system such that:*

- *the architecture is given by Algorithm 1 above;*
- *all the feedforward connections have weight* W*;*
- *all the recurrent connections have weight 1;*
- *all the hidden and output neurons have threshold* θ*;*
- *all neurons have activation functions* $h(\cdot)$*,*

where W*,* θ *and* $h(\cdot)$ *are computed as done for CILP in (d'Avila Garcez, Broda, and Gabbay 2002).*

Definition 2. *A* **Rule-based Neural Network System** *is a tuple* $\langle RNN, I, F, C_o, C_r\rangle$*, where*

- RNN *is a rule-based neural network as introduced in Definition 1;*
- I *is a set of rule names to be activated ad the beginning of the monitoring;*
- F *is a set of rule names that are required not to be activated at the end of the monitoring;*
- C_r *is a function that checks the output of the network for consistency;*
- C_o *is a function that checks consistency between the incoming observations and the rule-generated obligations.*

We compute weights, thresholds and activation functions of RNNs as in (d'Avila Garcez, Broda, and Gabbay 2002) in order to implement the following conditions:

- the input potential of a hidden neuron x_h can only exceed x_h's threshold (θ_{x_h}), activating x_h, when all the positive antecedents of rule x are assigned the truth value true while all the negative antecedents of x are assigned false;
- the input potential of an output neuron x_o can only exceed x_o's threshold (θ_{x_o}), activating x_o, when at least one hidden neuron y_h that is connected to x_o is activated.

The hidden neurons of a RNN have two additional features: threshold update and disjunct selection. The threshold update feature mirrors the activation of the respective rule. In RuleR, if a rule name occurs in the state generated by the body of the activated rules, in the next iteration that rule is active, that is, its condition can be checked and, if this is the case, its body is triggered. In RNN, after a hidden neuron fires, it sets its threshold to an arbitrarily high value (symbolically, $+\infty$). This will prevent the hidden neuron from firing again at the next iteration, as no input will be large enough to pass this threshold. Each positive rule-labelled output neuron x_o that gets triggered afterwards will then set the threshold of the respective hidden neuron (x_h, that is, the neuron with the same label) back to θ_{x_h}, so that this neuron can fire in the next iteration if the input signal is strong enough. Summarizing, hidden neurons do not fire (due to the high threshold) unless the respective output neuron fired at the previous iteration, just like RuleR rules will not be active unless the rule name occurred in the output status in the previous iteration.
Disjunct selection copes with the body of the rules: they are disjunctions of conjunctions of literals. It is worth noting that RuleR handles a frontier of possible states: set I may contain different initial states with the disjunctions in the body of the rules generating different states (although some may be pruned due to inconsistency). Note that, when several rules with disjunctive bodies are triggered, the disjuncts have to be selected and combined. To implement this, the network is made to reason hypothetically. For simplicity, we use a single state and select a disjunct from the body of each active rule at each iteration. This assumption of disjuncts allows a more compact representation in the neural network than representing the entire frontier would, but may require multiple runs over the same trace, since a trace may satisfy the rule system by making a wrong assumption of disjuncts. We will discuss this choice in the experiments. In particular, we will evaluate how, with some implementation adjustments, this choice has little impact on expressive power and performance. In contrast, if we were to implement a single-state frontier in the network, we would be able to monitor several traces at a time, since a batch can be encoded as a matrix, each line corresponding to a trace. This implementation decision may depend on the application at hand according to its theoretical maximum parallel speedup.

4 Monitoring

Given a RuleR rule system R, we can generate a rule-based neural network system $RNNS = \langle RNN, I, F, C_o, C_r\rangle$. This system can be used to check if traces comply with the temporal logic property encoded by R. A trace is a finite sequence of observations, $\tau = o_1, o_2, .., o_n$. The monitoring algorithm is described in Algorithm 2.
Lines (1 to 4) initialize the $RNNS$: the rules in the initial set are used to activate the corresponding rule-labelled input neurons and to set the threshold of the corresponding hidden neurons to its nominal value θ. Lines (5 to 16) describe the iterative part of the monitoring cycle: line (6) checks for observation consistency, as observation obligations from the

Algorithm 2 Monitoring algorithm

```
for all $x \in I$ do
    Feed 1 as input to $x_i$
    Set $\theta_{x_h}$ to $\theta$
end for
for $i = 1 \to n$ do
    if $\neg C_o$ then
        return false
    end if
    add $o_i$ to the input layer
    Feedforward activation
    Set the hidden neuron thresholds
    if $\neg C_r$ then
        return false
    end if
    Recurrent activation
end for
if $\exists\ x_o\ s.t.\ x \in F, x_o$ is active then
    return false
else
    return true
end if
```

previous iteration may not match the current actual observations. Any consistency check failure causes the termination of the algorithm with negative outcome. If the observation consistency check succeeds, the new observations are added to the network (i.e. feeding the inputs of the respective input neurons so that they are activated), the network is run feedforward and, for each active output neuron labelled with a positive rule name, the respective hidden neuron's threshold is set to its nominal value. Another consistency check is performed on the resulting state, as different rules may have activated output neurons with inconsistent labels. If this check succeeds, the output layer's output is propagated back, through the network's recurrent connections, to the input layer, and another iteration begins. Lines (17 to 21) check whether any forbidden rule is still active after the trace terminates (i.e. a stable state is found in the recurrent network): if this is the case, the trace does not satisfy the rule system, and the algorithm returns *false*. If, however, no forbidden rule is active, the trace satisfies the rule system and the outcome is *true*. Whether or not a rule r is activated can be checked in the final state of the network by checking the activation of the output-layer, r-labelled neuron r_o.

4.1 Semantics

In this section we give a formal account of the monitoring algorithm and an evaluation semantics over traces of observations. One key intuition is that, although we encode a rule system in a neural network, all neurons keep a symbolic interpretation, as the activation of a neuron is interpreted as an occurrence of that rule/observation name.

Definition 3. *A configuration γ for a RNN at time t is the set of active neurons in its output layer at t. We will decompose γ into γ_A and γ_O, the first one being the set of rule-labeled neurons in γ and the second being the set of observation-labeled neurons in γ.*

Definition 4. . *A set of literals L is said to hold in a configuration γ iff $(L \backslash O^{\pm}) \subseteq \gamma_A \ \wedge (L \backslash R^{\pm}) \subseteq \gamma_O$.*

Definition 5. *A set of observation literals L is said to match a configuration γ if and only if $L \cap \gamma$ is consistent and $\gamma \subseteq L$. The function $match$ performs such a check. A set of (matching) observations L can be added to an input-configuration γ by the function $addObs$ which activates the input neurons corresponding to literals in L.*

Definition 6. *A step between configurations, $\gamma \longrightarrow \gamma'$, holds iff $\forall x \in \gamma', \exists\ y \ \in \ \gamma_A\ s.t.\ \forall z \in C(y), z \in \gamma$.*
This means that every active neuron x in γ' has been activated by a hidden neuron y which was active in γ and had its condition satisfied, as all neurons in its condition belonged to γ as well.

Definition 7. *A RNN $\langle RNN, I, F, C_o, C_r \rangle$ accepts an observation trace $\tau = o_1, o_2, .., o_n$ if there exists a sequence of configurations $\gamma = \gamma_1, \gamma_2, .., \gamma_n$ such that:*

- $\exists i \in I\ s.t.\ i = \gamma_1$
- $\forall \gamma_{i:1}^{n-1}, match(o_i, \gamma_{iO}) \wedge addObs(\gamma_i, o_i) \longrightarrow \gamma_{i+1}$
- $\gamma_n \cap F = \emptyset$

A trace that is not accepted by a RNN (that is, for which there exists no accepting run) is said to violate *the RNN.*

Definition 8. *The language of the rule-based network $L(RNN)$ is the set of finite traces τ such that RNN accepts τ.*

Remark 1. *Above, we have redefined RuleR's semantics in term of active neurons, mirroring its definitions from literals to neural networks. Therefore, the RNN built upon a rule system R accepts the same language of R, as at any time the configuration in R (in terms of positive literals) corresponds to the configuration of RNN (represented by active neurons).*

5 Example

We now give an example of how a temporal logic formula can be transated into a rule system and into a RNNS; we then show an example of run over an observation trace. Suppose we start from the SNF formula:

$$\Box((a \wedge b) \ \Rightarrow \bigcirc(c\, U d))$$

Using the following abbreviations:

- $r_{until} = r_{cUd}$,
- $r_{gen} = r_{g,(a \wedge b) \Rightarrow \bigcirc(cUd)}$,
- $r_{link} = r_{(a \wedge b) \Rightarrow \bigcirc(cUd)}$

the resulting RuleR system $RS = \langle R, O, P, I, F \rangle$ becomes:

- $R = \{r_{until}, r_{gen}, r_{link}\}$
- $O = \{a, b, c, d\}$

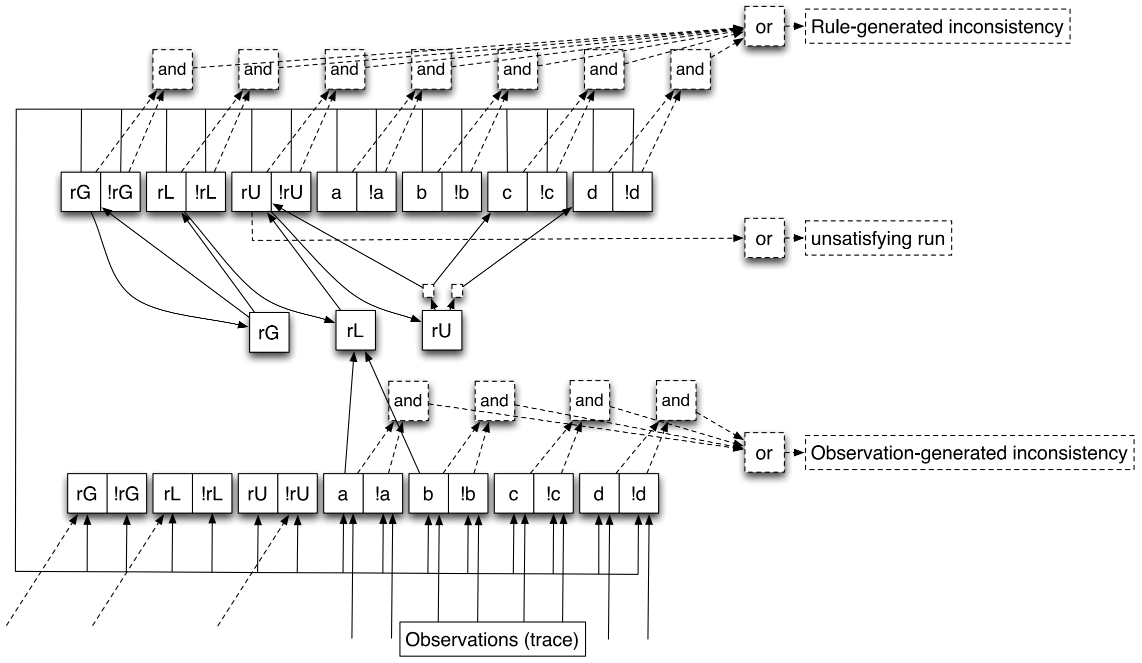

Figure 1: RNNS obtained from $RuleR$ system RS

- $P = \{r_{until} :\longrightarrow d \mid c, r_{until},$
  $r_{gen} :\longrightarrow r_{gen}, r_{link},$
  $r_{link} : a, b \longrightarrow r_{until}\}$
- $I = \{\{r_{gen}, r_{link}\}\}$
- $F = \{r_{until}\}$

Intuitively, r_{gen} *keeps* r_{link} *alive*, that is, it ensures that r_{link} is active at any step of the monitoring. r_{link} connects the *present-past* and *future* parts of the SNF formula: once a and b are observed at the same time, the antecedent of the implication holds, and the consequent has to be satisfied, so r_{link} triggers r_{until}. r_{until} reactivates itself as long as c is observed, and stops as soon as a d is observed. r_{until} being active means that the final condition of the until (that is, observing d) has not been fulfilled yet: this is why r_{until} appears in F.

The resulting RNNS is depicted in Figure 1. In the picture, dashed lines and boxes are component of the RNNS that do not belong to the neural network: consistency checks, initial value and forbidden rules. Note that the neural network is scarcely connected; for monitoring tasks, as no weight changes in the network is required, several neurons can be removed. The entire network is needed, though, if we were to perform learning. In this case, the network should be connected fully and initialized with small random weights, as done by CILP. A learning algorithm may then change the network's weights in the usual way (d'Avila Garcez, Broda, and Gabbay 2002). Learning is left as future work.

For the sake of presentation clarity, an accepting run of the RNNS over a trace is showed in Table 1. The table has to be read as follows: each row corresponds to an iteration; the *Rules* column lists the output states generated by the active rules in the previous round merged with the current observations (for the first row, this corresponds to the rules in I). The *Resulting state* column is the union of the bodies of the rules in the *Rules* column such that their conditions were satisfied. Since r_{until} is not active at the end of the run (bottom-right cell), the trace is accepted.

Table 1: Example of accepting run

#	**Obs.**	**Rules**	**Resulting state**
1	a,c	r_{gen}, r_{link}	r_{gen}, r_{link}
2	b,d	r_{gen}, r_{link},b,d	r_{gen}, r_{link}
3	a,b,c	r_{gen}, r_{link},a,b,c	$r_{gen}, r_{link}, r_{until}$
4	a,c	r_{gen}, r_{link},a,c	$r_{gen}, r_{link}, r_{until}$,c
5	b,c	r_{gen}, r_{link},b,c	r_{gen}, r_{link},d
6	a,d	r_{gen}, r_{link},a,d	r_{gen}, r_{link}

6 Experiments

We have implemented the $RNNS$ in Prolog and GNU Octave. Prolog was mostly used for the translation, as this phase requires parsing the RuleR rules; the Prolog code creates a RNNS in Octave and then matrices are used to perform monitoring through network activation. One big difference between RuleR and RNNS is that the first uses a breadth-first exploration strategy, while the latter uses a randomized depth-first search. Neither systems use backtracking: RuleR mantains a frontier of states, while RNNS randomly picks which node to explore and, in case of failure, starts again from the beginning. With some rule manipulation and implementation choices, we were able to steeply narrow the

branching factor, often generating completely deterministic RNNS.
For our tests, we used three custom-designed properties:

- $a \Rightarrow \bigcirc(true\, U b)$
- $a \wedge b \Rightarrow \bigcirc(c\, U(d \vee e))$
- $a \wedge b \Rightarrow \bigcirc((true\, U(c \wedge d)) \vee (e\, W false))$

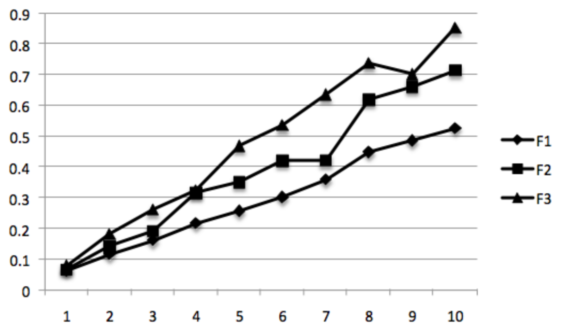

Figure 2: Scalability of time over trace size

We generated a $RNNS$ from each rule and fed it satisfying traces. Our goal was to analyze the impact of trace length on one hand, and property size on the other. Figure 2 shows the numerical results of a small batch of tests: on the X axis is the number of states of the trace (divided by ten: actual values span from 10 to 100); on the Y axis is the time (in seconds) required for the system to perform an accepting run over the trace. The three curves, namely F1, F2 and F3, represent three properties of increasing size. This example shows that, although $RNNS$ may require several runs over the same trace in order to reach an accepting run, the number of runs does not seem to impact much on performance, as the monitoring time seems to increase linearly with the trace size.

7 Future work

In the area of process mining, the key aspects are *model discovery*, *conformance checking* and *model enhancement* (van der Aalst et. al 2011). In many business process scenarios, even if a system is rigorously defined, its implementation may slightly deviate from the model: for instance, the protocol within an office may vary due to an unavaiable resource (a broken printer) or external constraints (a strike, a delay in office supplies delivery, etc.). The analysis of event logs can provide insight into how processes actually take place, and to what extent actual processes deviate from a normative process model (Goedertier et al. 2009). Having encoded a rule system in a neural network-based system, we conjecture that its intrinsic flexibility and noise tolerance will enable us to model this adaptation process through learning and verification from event logs in the $RNNS$.

8 Conclusions

This paper contains, to the best of our knowledge, the first implementation of a rule-based monitoring system in the neural-symbolic paradigm. We have provided an algorithm for translating RuleR rules into a neural network-based monitoring system and an explanation of how trace monitoring can be performed in the network. We have implemented our system and tested it on three benchmark examples which indicate that it should scale well to large-size properties even when traces have to be calculated repeatedly. This and the system's possibility of handling errors (noise) in the network, indicate em business process adaptation as a promising application area for $RNNS$. Next, we shall embed the system in a process mining iterative framework. To do this, we shall investigate learning strategies and knowledge extraction mechanisms, and analyze system performance, accuracy and scalability in this application area.

References

Barringer, H.; Rydeheard, D. E.; and Havelund, K. 2010. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.* 20(3):675–706.

Borges, R. V.; d'Avila Garcez, A. S.; Lamb, L. C.; and Nuseibeh, B. 2011. Learning to adapt requirements specifications of evolving systems. In *ICSE*, 856–859.

Borges, R. V.; d'Avila Garcez, A. S.; and Lamb, L. C. 2011. Learning and representing temporal knowledge in recurrent networks. *IEEE Transactions on Neural Networks* 22(12):2409–2421.

d'Avila Garcez, A. S., and Zaverucha, G. 1999. The connectionist inductive learning and logic programming system. *Appl. Intell.* 11(1):59–77.

d'Avila Garcez, A.; Broda, K.; and Gabbay, D. 2002. *Neural-Symbolic Learning Systems: Foundations and Applications*. Perspectives in Neural Computing. Springer.

Fisher, M. 1997. A normal form for temporal logic and its application in theorem-proving and execution. *Journal of Logic and Computation* 7:429–456.

Gabbay, D. M. 1987. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, 409–448.

Goedertier, S.; Martens, D.; Vanthienen, J.; and Baesens, B. 2009. Robust process discovery with artificial negative events. *Journal of Machine Learning Research* 10:1305–1340.

Hoelldobler, S., and Kalinke, Y. 1994. Towards a new massively parallel computational model for logic programming. In *ECAI94 workshop on Combining Symbolic and Connectioninst Processing*, 68–77.

Towell, G. G., and Shavlik, J. W. 1994. Knowledge-based artificial neural networks. *Artif. Intell.* 70(1-2):119–165.

van der Aalst et. al, W. M. P. 2011. Process mining manifesto. In *Business Process Management Workshops (1)*, 169–194.