Combining Runtime Verification and Property Adaptation through Neural-Symbolic Integration

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

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

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

Abstract

We propose a framework for combining runtime verification and learning in connectionist models. This framework is expected to offer an improved efficiency of the verification process through parallel computation, and a new mechanism of compliance with soft-constraints through learning (adaptation). The goal of business process adaptation is to discover, monitor and improve real processes by extracting knowledge from real-time event logs readily available in today's information systems. Within this wider framework, we have developed a run-time verification system for linear temporal logic, called RuleRunner, which we have designed explicitly with the goal of translating into a neural network for parallel computation and learning. In this paper, using results from neural-symbolic integration, we introduce the translation algorithm that enables the encoding of any RuleRunner specifications into standard CILP recurrent neural networks. The obtained network is shown to converge to the evaluation status of RuleRunner for each cell of the trace. Initial experimental results, also reported in this paper, indicate that the sparse version of the network representation scales better than the symbolic implementation of RuleRunner. The resulting system is a neural network capable of efficient runtime verification and offering well-known learning capabilities, which are being investigated. The framework will be applied to the adaptation of business processes capable of tolerating soft-violations occurring in real practice.

1 Introduction

Existing runtime verification systems are almost always built on the assumption that the specifications they are going to verify are fixed. This assumption is increasingly challenged in domains and tasks like the verification of compliance of business processes. In particular, it is relevant to distinguish two kind of violations: hard and soft. Hard violations match the classical definition of faults, i.e., errors that determine an unacceptable behaviour from the monitored system. Soft violations, on the other hand, are discrepancies between high-level directives and real implementation.

For example, in a bank, the MIFID regulation prescribes that before signing, the customer of a financial product should be informed adequately. This is a hard constraint which should not be violated. In contrast, a constraint coming from internal regulations, such as that some document must be sent in paper format rather than scanned, can be violated without major problems, if the scanned version is still compliant with laws. Thus, if a branch is considered successful (and is compliant with hard constraits) by substituting paper for scanned versions of a given document then the original specification may need to be revised.

Therefore, the problem of runtime verification must overcome the classical fault-detection schema, becoming a more flexible, fine-grained approach able to detect and report the hard violations, but also to integrate the soft violations in the encoded formal specification. A sketch of the desired system is shown in Figure 1.

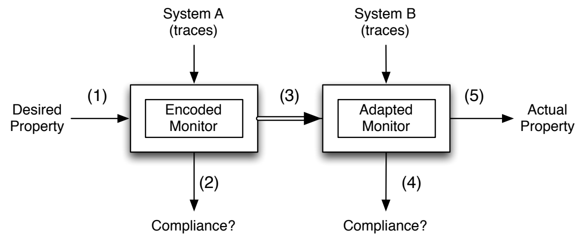

Figure 1: Sketch of the desired verification system

In this vision, the verification system should:

1. Encode the specified properties, for example in a formalism like LTL [Pnueli, 1977], and take as input traces from the observed system A.
2. Verify the compliance of system A's traces w.r.t the encoded property description, checking whether unacceptable normative violations occur.
3. If A exhibits only soft violations and its performance is better than other systems, learn its behaviour by observing it, merging A's ideal description and its actual behaviour in a new monitor.

4. Use the newly obtained monitor to verify the compliance of a second system, B. Intuitively, the idea is to verify how affine is the organisation of A and B, both concerning the normative directives and the actual process management.
5. Extract, from the adapted monitor, a new specification of the actual process management activity in A where the properties are revised according to the traces of A.

In this paper, we propose to use a neural network as the encoded monitor of Figure 1, focusing on the first two items: encoding and reasoning. We describe our rule-based runtime verification system, RuleRunner, and we introduce an algorithm to encode it in a recurrent neural network; the resulting system is a neural network capable of efficient runtime verification. We implemented all components and we obtained encouraging preliminary results in terms of performance. Furthermore, our system offers well-known learning capabilities, which are being investigated.

The paper is structured as follows: Section 2 introduces related work, Section 3 describes the RuleRunner system. Section 4 describes the neural encoding of a RuleRunner system. Section 5 discusses implementation and some performance results of the prototype, Section 6 ends the paper with conclusions and future work.

2 Background

2.1 Business Process Management

The IEEE Task Force on Process Mining aims to promote the topic of process mining. In the context of this task force, a group of more than 75 people involving more than 50 organisations created the Process Mining Manifesto. By defining a set of guiding principles and listing important challenges, this manifesto hopes to serve as a guide for software developers, scientists, consultants, business managers, and end-users. The goal is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs readily available in today's (information) systems. Therefore, process mining systems aim at closing the gap between the huge (and growing) amount of stored data about processes and activities and the need for making the business process management flexible and competitive for ever-evolving contexts.

The Business Process Management provides an application domain for our framework: when it comes to big companies (such as a bank), in some of their branches the task execution often differs from the rigid protocol enforcement, due to obstacles (from broken printers to strikes) or by adaptations to specific needs (dynamic resources reallocation): these are examples of soft violations. On the other hand, law infringements concerning security and privacy issues are hard violations. For instance, having the customer of a financial product sign a contract without being adequately informed is a (hard) violation of the MIFID regulation[1].

[1]Markets in Financial Instruments Directive 2004/39/EC (link)

Thus, if a branch considered successful and compliant with hard constraints, substituted paper versions of a given documents by scanned ones despite the original specifications, such specifications may be revised. The source of the modification should be the traces of the activity of the successful branch, which can be fed as input to a machine learning system.

2.2 Runtime Verification

Runtime verification (RV) of a given correctness property ϕ (often formulated in linear temporal logic LTL [Pnueli, 1977]) aims at determining the semantics of ϕ while executing the system under scrutiny; a monitor is defined as a device that reads a finite trace and yields a certain verdict [Leucker and Schallhart, 2009]. A trace is a sequence of cells, which in turn are lists of observations occurring in a given discrete span of time. Runtime verification may work on finite (terminated) traces, finite but continuously expanding traces, or on prefixes of infinite traces. A monitor may control the current execution of a system (online) or analyse a recorded set of finite executions (offline). There are many semantics for finite traces: FLTL [Lichtenstein *et al.*, 1985], RVLTL [Bauer *et al.*, 2007], LTL3 [Bauer *et al.*, 2006], LTL [Eisner *et al.*, 2003] just to name some. Since LTL semantics is based on infinite behaviours, these semantics aim to close the gap between properties specifying infinite behaviours and finite traces. There exist several RV systems [Leucker and Schallhart, 2009], and they can be clustered in three main approaches, based respectively on rewriting, automata and rules. In this paper, we focus on rule-based system, due to their similarity with neural networks.

2.3 Neural-Symbolic Integration

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 and Zaverucha, 1999]. CILP builds upon KBANN and [Hoelldobler and Kalinke, 1994] to provide a sound theoretical foundation for reasoning in artificial neural networks with learning capabilities. The general framework of a neural symbolic system is composed of 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 particular, rules are mapped onto hidden neurons, the preconditions of rules onto input neurons and the conclusion of the rules onto output neurons. The weighs are then adjusted to express the dependence among all these elements [Garcez *et al.*, 2002].

3 RuleRunner

RuleRunner is a rule-based online monitor observing finite but expanding traces and returning an FLTL verdict. As it scans the trace, RuleRunner mantains a state composed of rule names (for reactivating the rules), observations and formulae evaluations.

Given a finite set of observations O and a LTL formula ϕ over (a subset of) O, RuleRunner has a state composed of observations ($o \in O$), rule names ($R[\phi]s$) and truth evaluations ($[\phi]V$); $V \in \{T, F, ?\}$ is a truth value and s, called *suffix*, is used to identify formulae. Due to the lack of space, we will omit the technical details of RuleRunner, [Perotti, 2013] focusing on how the monitoring task is performed and how the rules are structured: this will be relevant in the next subsection. The state evolves according to rules: RuleRunner is composed of evaluation and reactivation rules. Evaluation rules are shaped like

$$R[\phi], [\psi^1]V, \ldots, [\psi^n]V, \rightarrow [\phi]V$$

and, intuitively, their role is to compute the truth value of a formula ϕ under verification, given the truth values of its direct subformulae ψ^i. Reactivation rules are shaped like

$$[\phi]? \rightarrow R[\phi], R[\psi^1], \ldots, R[\psi^n]$$

and, if one formula is evaluated to undecided, that formula (together with its subformulae) are scheduled to be monitored again in the next cell of the trace. This concept of rule reactivation was firstly described in [Barringer *et al.*, 2010].

A RuleRunner rule system RR is defined as $\langle R_E, R_R, S\rangle$ where R_E anr R_R are, respectively, the sets of evaluation and reactivation rules, and S is the initial state. The general algorithm for creating and running a RR is described in Algorithm 1:

Algorithm 1 Preprocessing and Monitoring Cycle

```
 1: Parse the LTL formula in a tree
 2: Generate evaluation rules, reactivation rules and the
    initial state
 3: while new observations exist do
 4:     Add observations to state
 5:     Compute truth values using evaluation rules
 6:     Compute next state using reactivation rules
 7:     if state contains SUCCESS or FAILURE then
 8:         return return SUCCESS or FAILURE
    respectively
 9:     end if
10: end while
```

In a nutshell, RuleRunner's behaviour is the following: in the preprocessing phase, RuleRunner encodes an LTL formula in a rule system. The rule system verifies the compliance of a trace w.r.t. the encoded property by entering a monitoring loop, composed of observing a new cell of the trace and computing the truth value of the property in the given cell. If the property is irrevocably satisfied or falsified in the current cell, RuleRunner outputs a binary verdict. If this is not the case, another monitoring iteration is entered, and -like in [Barringer *et al.*, 2010]- undecided formulae trigger the reactivation of the corresponding monitoring rule. FLTL semantics guarantees that, if the trace ends, the verdict in the last cell of the trace is binary.

It is worth stressing how RuleRunner's approach is *bottom-up*, forwarding truth values from mere observations to the global property. RuleRunner does not keep a [multi]set of alternatives, as it is rooted in matching the encoding of the formula with the actual observations, computing the unique truth value of every subformula of the property, and carrying along a single state composed of certain information.

RuleRunner provides rich information about the 's' of a property: in any iteration the state describes which subformulae are under monitoring and what the truth value is; when the monitoring ends, the state itself explains why the property was verified/falsified.

One minimal example that shows all functionalities of RuleRunner is $\Diamond a$: we will use it as a working example throughout the paper. In the next example, we consider the RuleRunner system encoding $\Diamond a$ and its evolution over the trace $[b - a - b, END]$.

The desired behaviour of the monitoring is the following:

- In the first cell, b is observed and ignored, while non observing a makes a false. Since the end of the trace has not been reached, $\Diamond a$ is undecided, and the initial state is repeated in the second cell. Note that in the general case the initial state is not identically recomputed in the following cells.
- In the second cell, a is observed, thus satisfying, in cascade, a and $\Diamond a$; since the main formula has been satisfied, the monitoring can stop in the current cell reporting a success.

RuleRunner creates the following rule system:

EVALUATION RULES

- $R[a]$, a is observed $\rightarrow [a]T$
- $R[a]$, a is not observed $\rightarrow [a]F$
- $R[\Diamond a]$, $[a]T \rightarrow [\Diamond a]T$
- $R[\Diamond a]$, $[a]? \rightarrow [\Diamond a]?P$
- $R[\Diamond a]$, $[a]F \rightarrow [\Diamond a]?P$
- $[\Diamond a]?, [END] \rightarrow [\Diamond a]F$
- $[\Diamond a]?, \sim[END] \rightarrow [\Diamond a]?$
- $[\Diamond a]T \rightarrow SUCCESS$
- $[\Diamond a]F \rightarrow FAILURE$

REACTIVATION RULES

- $[\Diamond a]? \rightarrow R[a], R[\Diamond a]$

INITIAL STATE $R[a], R[\Diamond a]$

EVOLUTION OVER $[b-a-b, END]$

state	$R[a], R[\Diamond a]$
+ obs	$R[a], R[\Diamond a], b$
eval	$[a]F, [\Diamond a]?P, [\Diamond a]?$
react	$R[a], R[\Diamond a]$
state	$R[a], R[\Diamond a]$
+ obs	$R[a], R[\Diamond a], a$
eval	$[a]T, [\Diamond a]T, SUCCESS$
STOP	PROPERTY SATISFIED

It is easy to see that RuleRunner's behaviour matches the desired one, [Perotti, 2013] provides a detailed description of RuleRunner's behaviour.

4 Neural Encoding

The first step of the neural encoding is the translation of a RuleRunner system into an equivalent logic program (Algorithm 2).

Algorithm 2 From RuleRunner to Logic Programs

```
 1: function RR2LP($\phi$)
 2:     Create $RR = \langle R_E, R_R, S\rangle$ encoding $\phi$
 3:     Create an empty logic program $LP$                          $\triangleright C_E$
 4:     for all $R[\phi], [\psi^1]V, \dots, [\psi^n]V, \to [\phi]V \in R_E$ do
 5:         $LP \leftarrow LP \cup [\phi]V$ :-$\sim[U], R[\phi], [\psi^1]V, \dots, [\psi^n]V$
 6:     end for                                                     $\triangleright C_P$
 7:     for all $o \in R_E \cup R_R$ do
 8:         $LP \leftarrow LP \cup$ $o$ :- $\sim[U], o$
 9:     end for
10:     for all $R[\phi] \in R_E \cup R_R$ do
11:         $LP \leftarrow LP \cup$ $R[\phi]$ :- $\sim[U], R[\phi]$
12:     end for                                                     $\triangleright C_R$
13:     for all $[\phi]? \to R[\phi], R[\psi^1], \dots, R[\psi^n] \in R_R$ do
14:         $LP \leftarrow xLP \cup$ $R[\phi]$ :- $[U], [\phi]?$
15:         for all $R[\psi^i]$ do
16:             $LP \leftarrow LP \cup$ $R[\psi^i]$ :- $[U], [\phi]?$
17:         end for
18:     end forreturn $LP$
19: end function
```

The algorithm creates a single logic program LP; however, for the sake of explanation, we distinguish three kinds of clauses: evaluation, reactivation and persistence (marked as C_E, C_R, C_P in Algorithm 2). Intuitively, evaluation and reactivation clauses (in LP) mirror, respectively, evaluation and reactivation rules in RuleRunner. Persistence clauses are used to 'remember' observations and active rules by explicit re-generation: these clauses follow the pattern x :- $\sim[UPDATE], x$ (shortened to *[U]* in the algorithm).

Evaluation clauses are obtained from evaluation rules by adding one extra literal in the body, $\sim[UPDATE]$. The reactivation rules are split into several reactivation clauses, one for each literal in the head of the rule; $[UPDATE]$ is added in the body of all these rules. Finally, for all observations and truth evaluations in the rules, a persistence clause is added.

RuleRunner's monitor loop fires the evaluation and reactivation rules in an alternate fashion. We simulate that by introducing the $[UPDATE]$ literal and using it as a switch: when $[UPDATE]$ holds, only reactivation clauses can hold, while when it does not all reactivation rules are inhibited and evaluation and persistence clauses are potentially active. RuleRunner iteratively builds a state, going through partial solutions; in the same way, some sort of clamping is necessary for the partial results to be remembered by the LP. We achieve that by adding persistence clauses: as long as $[UPDATE]$ does not hold, all observations and rule names are re-obtained at each iteration of the logic program. Another way to achieve this persistence is to add the desired observations/rule names as facts: we opted for the former because persistence clauses have a standard structure, while adding facts to the LP correspond to clamping neurons in a neural network.

In our working example, the rule system encoding $\Diamond a$ is translated into the following logic program:

EVALUATION CLAUSES

- $[a]T$:- $\sim[UPDATE], R[a], a$
- $[a]F$:- $\sim[UPDATE], R[a], \sim a$
- $[\Diamond a]T$:- $\sim[UPDATE], R[\Diamond a], [a]T$
- $[\Diamond a]?P$:- $\sim[UPDATE], R[\Diamond a], [a]?$
- $[\Diamond a]?P$:- $\sim[UPDATE], R[\Diamond a], [a]F$
- $[\Diamond a]?$:- $\sim[UPDATE], [\Diamond a]?, \sim[\Diamond a]T, \sim END$
- $[\Diamond a]F$:- $\sim[UPDATE], [\Diamond a]?P, \sim[\Diamond a]T, END$
- $[SUCCESS]$:- $\sim[UPDATE], [\Diamond a]T$
- $[FAILURE]$:- $\sim[UPDATE], [\Diamond a]F$

PERSISTENCE CLAUSES

- a :- $\sim[UPDATE], a$
- $R[a]$:- $\sim[UPDATE], R[a]$
- $R[\Diamond a]$:- $\sim[UPDATE], R[\Diamond a]$

REACTIVATION CLAUSES

- $R[a]$:- $[UPDATE], [a]?$
- $R[a]$:- $[UPDATE], [\Diamond a]?$
- $R[\Diamond a]$:- $[UPDATE], [\Diamond a]?$

Summarising, we start from a formal property ϕ expressed as an LTL formula, we compute a RuleRunner rule system RR_ϕ monitoring that formula, and then we encode the rule set in a logic program LP_ϕ. By doing so, we can exploit the $CILP$ algorithm [d'Avila Garcez and Zaverucha, 1999] to translate the logic program into an equivalent neural network NN_ϕ.

Continuing with the working example, the neural network encoding a monitor for $\Diamond a$ is visualised in Figure 2. The input and output layers include neurons whose labels correspond to atoms in the logic program (and the rule system); each hidden neuron corresponds to one clause in the logic program. Active neurons are filled in grey: in Figure

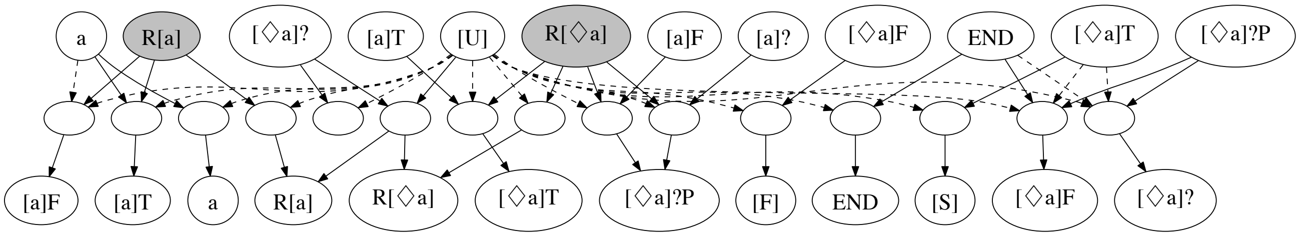

Figure 2: Neural network for monitoring $\Diamond a$

2, the initial state $(R[a], R[\Diamond a])$ is depicted. Solid lines represent connection with positive weights, dashed lines represent negative weights. For instance, the leftmost hidden neuron has ingoing connections from a (negative weight), $R[a]$, and $[U]$ (negative weight) and it has an outgoing connection towards $[a]F$; this corresponds to the clause $[a]F$:- $a, \sim a, R[a], \sim[U]$ (which is the second clause in the LP in the previous page).

Reasoning: The general algorithm for neural monitoring is described in Algorithm 3:

Algorithm 3 Preprocessing and Monitoring Cycle in NN_ϕ

```
 1: function NN-MONITOR($\phi$,trace t)
 2:     Create $RR = \langle R_R, R_E, S \rangle$ encoding $\phi$ (RuleRunner)
 3:     Rewrite $RR_\phi$ into $LP_\phi$ (Algorithm 2)
 4:     Rewrite $LP_\phi$ into $NP_\phi$ ($CILP$)
 5:     Add $S$ to the input layer
 6:     while new observations exist in t do
 7:         Add the new observations to the input layer
 8:         Let the network converge
 9:         if $S$ contains SUCCESS (resp.FAILURE) then
10:             return return SUCCESS (resp.FAILURE)
11:         end if
12:         Add $UPDATE$ to the input layer
13:         Fire the network once
14:     end while
15: end function
```

Adding x to a given layer means activating the neuron corresponding to x in that layer. In terms logic programming, this would correspond to adding the fact x to the program.

It is worth comparing how, from an operational point of view, a RuleRunner system (RR_ϕ) and its neural encoding (NN_ϕ) carry out the monitoring task. In each iteration of the main loop, RuleRunner goes through the list of evaluation rules, adding the result of each active rule to the state. When the end of the evaluation rules list is reached, the reactivation rules are allowed to fire, collecting the output in a new state. In the neural network the alternating of evaluation and reactivation is achieved by means of the $[UPDATE]$ neuron, which acts as a switch. In the evaluation phase, all evaluation rules are fired in parallel until convergence.

5 Implementation and Initial Results

We implemented RuleRunner[2] and the neural modules. Rulerunner is implemented in Java; in order to maximise the performances of the neural encoding, we experimented with several implementations, focusing on matrices manipulation.

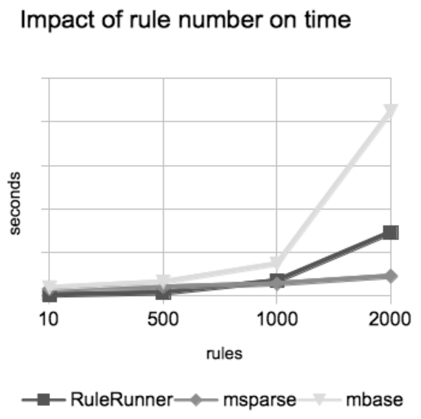

Figure 3: Compared performances

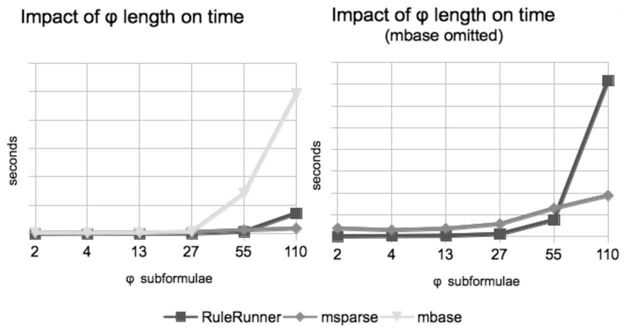

Figure 4: Compared performances

One key point in understanding how to fully exploit the parallelism of neural network models (and therefore optimise the performance of the neural encoding of RuleRunner)

[2]*www.di.unito.it/~perotti/RV13.jnlp*

is that, even if the number of rules is large, each rule has a constant number of literals in the body, and each literal appears in (at most) a constant number of rules. Therefore, the weight matrices are very sparse.

We tested RuleRunner's performance on a number of tests and compared the results with two Matlab implementations, m_base and m_sparse: in the latter, we treated all matrices (activations, thresholds, weights) as sparse. It is worth stressing that, since we compared two prototypes implemented in different programming languages (Java and Matlab), it would not be fair to compare their performances in absolute terms; we are, instead, interested in scalability and asymptotic analysis.

These preliminary experiments show how m_base is outperformed by RuleRunner, but m_sparse scales better than both. Figure 3 shows the impact of increasing the rule numbers, in Figure 4 the size of the encoded formula is increased. In all cases the Y axis reports the average time required to process 10000 cells of randomly-generated traces.

6 Conclusions

With the goal of making the concept of compliance flexible and dynamic, we tackled the challenge of developing a framework to rigidly verify a system's compliance with a model, modelling at the same time what actually takes place in terms of process management, making it exploitable by other systems. We chose Neural-Symbolic Integration as the underlying paradigm and rule-based verification system as a bridge from the (symbolic) area of Runtime Verification and machine learning through neural networks.

As a first contribution, this paper provides a methodology for performing runtime verification within a neural network, encoding a novel rule system in a logic program and then in a recurrent neural network; we developed a prototype for our system, observing that an implementation based on sparse matrices shows better performances than RuleRunner, both in absolute terms and from the point of view of scalability. Secondly, our approach seeks to reduce the gap between Runtime Verification, Artificial Intelligence and Business Process Management, proposing a system able to perform RV tasks in a model with intrinsic learning features.

Future work involve the exploitation of standard and ad-hoc learning strategies, in relation with sequence and reinforcement learning: the goal is to analyse the adaptation capability of the model in order to integrate formal properties, encoded in the network, with the actual behaviour of the observed system, by means of feeding traces to the network as inductive learning examples.

References

[Barringer *et al.*, 2010] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

[Bauer *et al.*, 2006] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[Bauer *et al.*, 2007] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007.

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

[Eisner *et al.*, 2003] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV'03*, pages 27–39, 2003.

[Garcez *et al.*, 2002] Artur S. d'Avila Garcez, Dov M. Gabbay, and Krysia B. Broda. *Neural-Symbolic Learning System: Foundations and Applications*. Springer-Verlag New York, Inc., 2002.

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

[Leucker and Schallhart, 2009] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[Lichtenstein *et al.*, 1985] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[Perotti, 2013] Alan Perotti. Rulerunner technical report. 2013. arXiv:1306.0810.

[Pnueli, 1977] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

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