Scalable Process Monitoring Through Rules and Neural Networks

Alan Perotti¹, Guido Boella¹, and Artur d'Avila Garcez²

¹ University of Turin, Italy
 ² City University London, United Kingdom

Abstract. In this paper we introduce RuleRunner, a Runtime Verification system for monitoring LTL properties over finite traces. By exploiting results from the Neural-Symbolic Integration area, a RuleRunner monitor can be encoded in a recurrent neural network. The results show that neural networks can perform real-time runtime verification and techniques of parallel computing can be applied to improve the performance in terms of scalability. Furthermore, our framework allows for property adaptation by using a standard neural network learning algorithm.

1 Introduction

A trend in high-performance computation is parallel computing, and the field of neural networks has been showing impressive improvements in performance, especially with the use of GPU accelerators. For example, in 2012 Google used approximately 1,000 CPU-based servers, or 16,000 CPU cores, to develop its neural network (1.7 billion parameters), which taught itself to recognise cats in a series of YouTube videos [13]. One year later, the Stanford team created an equally large network with only three servers using NVIDIA GPUs to accelerate the processing of the big data generated by the network [7]. Our first Research Question is the following:

Is it possible to use the connectionist paradigm and parallel computing techniques in order to improve the performance of a runtime verification system?

Neural networks are also known to offer intrinsic learning capabilities [11], and many fields using verification also involve learning-based tasks. For instance, Business Process (BP) Management [19] attempts to continuously improve business effectiveness and efficiency, typically relying on Business Process Management Systems (BPMS). A BPMS allows to automatically execute a BP according to its process schema, i.e., to a formalised model in which the actions to be performed and the control flow relations to be respected among them are specified. However, BP optimisation may ask the enterprise to be able to flexibly change and adapt such a predefined process schema, in response to expected situations (e.g., new laws, reengineering efforts, *concept drift* [19]) as well as to unanticipated exceptions and problems in the operating environment (e.g., emergencies) [12]. Our second Research Question is therefore:

Can one develop a single framework that combines efficient monitoring, possibly through parallel computing techniques, with property adaptation through learning?

In this paper, we answer both Research Questions positively. We present our runtime

monitoring system: RuleRunner creates and maintains a single, detailed state, avoiding the *possible worlds* branching structure typical of tableaux-based formulations. Furthermore, RuleRunner is designed as a set of rules that can be fired in parallel, rather than in a strict sequence. We show how a RuleRunner system can be translated into an equivalent logic program, in order to exploit results from the Neural-Symbolic Integration area [8] to encode it in a recurrent neural network. We validate our framework by comparing three implementations based on neural networks, observing how sparse form and GPU computing improve the performance, respectively, in terms of absolute speed and scalability. We show how in our framework learning corresponds to adapt the encoded property according to the observed behaviour of a system, but the application of learning strategies and approaches is outside the scope of this paper.

As an example, suppose several violations (signalled by an access control monitor) coming from a given area of a LAN turn out to be false positives [6]. The reasonable reaction is to relax the encoded property it in order to correctly identify the transactions from the given area as acceptable. In this way, the security system still encodes the given property, but reducing the number of false positives makes the system more efficient.

This paper is structured as follows: Section 2 introduces background and related work and Section 3 analyses our rule-based system, RuleRunner. Section 4 discusses the translation of our rule system in a neural network; Section 5 shows experimental results and a motivational example for parallel monitoring and property adaptation; Section 6 ends the paper with closing remarks and directions for future work.

2 Background and Related Work

2.1 Runtime Verification

Runtime Verification (RV) relates an observed system with a formal property ϕ specifying some desired behaviour. An RV module, or monitor, is defined as a device that reads a trace and yields a certain verdict [14]. 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), finite but continuously expanding, or on prefixes of infinite traces. While LTL is a standard semantic for infinite traces [17], there are many semantics for finite traces: FLTL [15], RVLTL [3], LTL3 [4], LTL \pm [10] just to name some. Since the standard LTL semantics is based on infinite behaviours, the issue is to close the gap between properties specifying infinite behaviours and finite traces. In particular, FLTL differs from LTL as it offers two *next* operators $(X, \overline{X} \text{ in } [3], X, W \text{ in this paper})$, called respectively strong and weak next. Intuitively, the strong (and standard) X operator is used to express with $X\phi$ that a next state must exist and that this next state has to satisfy property ϕ . In contrast, the weak W operator in $W\phi$ says that if there is a next state, then this next state has to satisfy the property ϕ . More formally, let $u = a_0 \dots a_{n-1}$ denote a finite trace of length n. The truth value of an FLTL formula ψ (either $X\phi$ or $W\phi$) w.r.t. u at position i < n, denoted by $[u, i \models \psi]$, is an element of \mathbb{B} and is defined as follows:

$$[u, i \models X\phi] = \begin{cases} [u, i+1 \models \phi], & \text{if } i+1 < n \\ \bot, & \text{otherwise} \end{cases} \quad [u, i \models W\phi] = \begin{cases} [u, i+1 \models \phi], & \text{if } i+1 < n \\ \top, & \text{otherwise} \end{cases}$$

While RVLTL and LTL3 have been proven to hold interesting properties w.r.t. FLTL (see [3]), we selected FLTL as we think it captures a more intuitive semantics when

2

dealing with finite traces. Suppose to monitor $\phi = \Box a$ over a trace t, where a is observed in all cells: we have that $[t \models \phi]$ equals, respectively, \top in FLTL, ? in LTL3, and T^p in RVLTL. If t is seen as a prefix of a longer trace $t\sigma$, then LTL3 and RVLTL provide valuable information about how ϕ could be evaluated over σ . But if t is a conclusive, self-contained trace (e.g. a daily set of transactions), then the FLTL semantics captures the intuitive positive answer to the query *does a always hold in this trace*? Several RV systems have been developed, and they can be clustered in three main approaches, based respectively on rewriting, automata and rules [14]. Within rule based approaches, RuleR [2] uses an original approach. It copes with the temporal dimension by introducing rules which may reactivate themselves in later stages of the reasoning. RuleRunner is inspired by this powerful idea, and in particular by the striking similarity of triggering rules and activating neurons. While various formalisms can be encoded

in a RuleR monitor, we focus on FLTL and extend RuleR to allow structures of a fixed size that can be implemented in efficient networks. The next section will describe the difference in the two approaches in more detail.

2.2 Neural networks and Neural-symbolic Integration

An artificial Neural Network (NN, [11]) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. Within the umbrella term of *connectionist approach*, many NN models have been developed, varying from those with only one or two layers of single direction logic, to complicated models with multi-input and many-directional feedback loops and layers. On the whole, these systems use algorithms in their programming to determine control and organisation of their functions. What they do have in common, however, is the principle of non-linear, distributed, parallel and local processing and adaptation.

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 [18], 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 [8] (example in Fig. 1). CILP's Translation Algorithm maps a general logic program P into a single-hidden-layer neural network N such that N computes the least fixed-point of P. 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 weights are then adjusted to express the dependence among all these elements. The obtained network implements a massively parallel model for Logic Programming, and it can perform inductive learning from examples, by means of standard learning strategies.



Fig. 1: Logic program P as a network N (CILP); (a) shows a simple logic program P. Applying CILP to P generates a NN N as depicted in (c). The input-outputs behaviour in P and N is shown in (b) and (d) respectively. For instance, if P is presented with the fact A, the first two clauses will produce A, B (second line in (b)). Similarly, if the input neuron representing A is activated in N, the signal propagation will activate the first two hidden neurons and both output neurons (second line in (d)).

3 The RuleRunner Rule System

RuleRunner is a rule-based online monitor observing finite but expanding traces and returning an FLTL verdict. RuleRunner accepts formulae ϕ generated by the grammar:

 $\phi ::= true \mid a \mid !a \mid \phi \lor \phi \mid \phi \land \phi \mid \phi U \phi \mid X \phi \mid W \phi \mid \Diamond \phi \mid \Box \phi \mid END$

a is treated as an atom and corresponds to a single observation in the trace. We assume, without loss of generality, that temporal formulae are in negation normal form (NNF), e.g., negation operators pushed inwards to propositional literals and cancellations applied. W is the weak next operator. *END* is a special character that is added to the last cell of a trace to mark the end of the input stream.

Algorithm 1 RuleRunner monitoring (abstract)			
1: fun	ction RR-MONITORING(ϕ , trace t)		
3:	while new cells exist in t do		
4:	Observe the current cell		
5:	Compute truth values of ϕ in the current cell of t	▷ Evaluation rules	
0: 7.	if ϕ is verified or falsified then		
8:	end if		
9:	Set up the monitor for the next cell in t	▷ Reactivation rules	
10:	end while		
11: end function			

Given an FLTL formula ϕ and a trace t, Algorithm 1 provides an abstract description of the creation and runtime behaviour of a RuleRunner system monitoring ϕ over t. At first, a monitor encoding ϕ is computed. Second, the monitor enters the verification loop, composed by 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 (because the ϕ refers to cells ahead in the trace), the system shifts to the following cell and enters another monitoring iteration. The FLTL semantics guarantees that, if the trace ends, the verdict in the last cell of the trace is binary. RuleRunner is a runtime monitor, as it analyses one cell at a time and never needs to store past cells in memory nor peek future ones. **Definition 1.** A RuleRunner system is a tuple $\langle R_E, R_R, S \rangle$, where R_E (evaluation rules) and R_R (reactivation rules) are rule sets, and S (for state) is a set of active rules, observations and truth evaluations.

Given a finite set of observations O and an FLTL formula ϕ over (a subset of) O, a state S is a set of observations ($o \in O$), rule names ($R[\psi]$) and truth evaluations ($[\psi]V$); $V \in \{T, F, ?\}$ is a truth value. A rule name $R[\psi]$ in S means that the logical formula ψ is under scrutiny, while a truth evaluation $[\psi]V$ means that the logical formula ψ currently has the truth value V. The third truth value, ? (*undecided*), means that it is impossible to give a binary verdict in the current cell.

Evaluation rules follow the pattern $R[\phi], [\psi^1]V, \dots, [\psi^n]V, \to [\phi]V$ and their role is to compute the truth value of a formula ϕ under verification, given the truth values of its direct subformulae ψ^i (line 5 in Algorithm 1). For instance, $R[\Diamond \psi], [\psi]T \to [\Diamond \psi]T$ reads as *if* $\Diamond a$ *is being monitored and* ψ *holds, then* $\Diamond \psi$ *is true.*

Reactivation rules follow the pattern $[\phi]? \to R[\phi], R[\psi^1], \ldots, R[\psi^n]$ and the meaning is that if one formula is evaluated to undecided, that formula (together with its subformulae) is scheduled to be monitored again in the next cell of the trace (line 9 in Algorithm 1). For instance, $[\Diamond \psi]? \to R[\Diamond \psi], R[\psi]$ means that if $\Diamond \psi$ was not irrevocably verified nor falsified in the current cell of the trace, both ψ and $\Diamond \psi$ will be monitored again in the next cell.

A RuleRunner feature is that rules never involve disjunctions. In RuleR, for instance, the simple formula $\Diamond a$ is mapped to the rule $R_{\Diamond a} : \longrightarrow a \mid R_{\Diamond a}$ and its meaning, intuitively, is that, if $\Diamond a$ has to be verified, either a is observed (thus satisfying the property) or the whole formula will be checked again (in the next cell of the trace). The same formula corresponds to the following set of rules in RuleRunner:

$R[\Diamond a], [a]T \to [\Diamond a]T$	$R[\Diamond a], [a]?, END \to [\Diamond a]F$
$R[\Diamond a], [a]? \to [\Diamond a]?$	$[\Diamond a]? \to R[a], R[\Diamond a]$
$R[\Diamond a], [a]F \to [\Diamond a]?$	

The disjunction in the body of the RuleR rule corresponds to the additional constraints in the head of the RuleRunner rules. Therefore, where RuleR generates a set of alternative hypotheses and later matches them with actual observations, RuleRunner maintains a detailed state of exact information. This is achieved by means of evaluation tables: three-valued truth tables (as introduced by Lukasiewitz [16]) annotated with *qualifiers*. Each evaluation rule for ϕ corresponds to a single cell of the evaluation table for the main operator of ϕ ; a *qualifier* is a subscript letter providing additional information to ? truth values. Figure 2 gives the example for disjunction. Qualifiers (B, L, R in this case) are used to store and propagate detailed information about the verification status of formulae. For instance, if ϕ is undecided and ψ is false when monitoring $\phi \lor \psi$ (highlighted cell in Figure 2), ?_L means that the disjunction is undecided, but that its future verification state will depend on the truth value of the Left disjunct. Note, in fact, how \lor_L is a unary operator. An example for this is monitoring $\Diamond b \lor a$ against a cell including only c: a is false, $\Diamond b$ is undecided (as b may be observed in the future), and the whole disjunction will be verified/falsified in the following cells depending on $\Diamond b$ only.



Fig. 2: Truth table (left) and evaluation tables (right) for \lor



Fig. 3: Complete set of evaluation tables

The complete set of evaluation tables is reported in Fig. 3, while the generation of evaluation and reactivation rules is summarised in Algorithm 2. The algorithm parses ϕ in a tree and visits the parsing tree in post-order. The system is built incrementally, starting from the system(s) returned by the recursive call(s). If ϕ is an observation (or its negation), an initial system is created, including two evaluation rules (as the observation may or may not occur), no reactivation rules and the single $R[\phi]$ as initial state. If ϕ is a conjunction or disjunction, the two systems of the subformulae are merged, and the conjunction/disjunction evaluation rules, reactivation rule and initial activation are added. The computations are the same if the main operator is U, but the reactivation rule will have to reactivate the monitoring of the two subformulae; in particular, U_A denotes the standard *until* operator, while U_B is the particular case where the ψ failed and the *until* operator cannot be trivially satisfied anymore. Formulae with X or W as main operator go through two phases: first, the formula is evaluated to undecided, as the truth value can't be computed until the next cell is accessed. Special evaluation rules force the truth value to false (for X) or true (for W) if no next cell exists. Then, at the next iteration, the reactivation rule triggers the subformula: this means that if $X\phi$ is monitored in cell i, ϕ is monitored in cell i + 1. ϕ is then monitored independently, and the $X\phi$ (or $W\phi$) rule enters a 'monitoring state' (suffix M in the table), simply mirroring ϕ truth value and self-reactivating. The evaluation of $\Box \phi$ is false (undecided) when ϕ is false (undecided); it is also undecided when ϕ holds (as $\Box \phi$ can never be true before the end of the trace), but the K suffix indicates when, at the end of the trace, an undecided \Box can be evaluated to true. Finally, $\Diamond \phi$ constantly reactivates itself and its subformula ϕ , unless ϕ is verified at runtime (causing $\Diamond \phi$ to hold), the trace ends ($\Diamond \phi$ fails).

Algorithm 2 Generation of rules

```
1: function INITIALISE(\phi)
 2:
                op \leftarrow main operator
                                                                                                                                                                              ▷ Apply recursively to subformula(e)
               \begin{array}{l} \text{if } op \in \{\Box, \Diamond, X, W\} \text{ then} \\ \langle R_E^1, R_R^1, S^1 \rangle \leftarrow \text{Initialise}(\psi^1) \end{array}
 3:
 4:
                        R_E \leftarrow R_E^1;
 5:
 6:
                        R_R^- \leftarrow R_R^1;
                else if op \in \{\vee, \wedge, U\} then

\langle R_E^1, R_R^1, S^1 \rangle \leftarrow \text{Initialise}(\psi^1)

\langle R_E^2, R_R^2, S^2 \rangle \leftarrow \text{Initialise}(\psi^2)
 7:
8:
 9:
10:
                          R_E \leftarrow R_E^1 \cup R_E^2; R_R \leftarrow R_R^1 \cup R_R^2;
11:
                   else
12:
13:
                          R_E \leftarrow \emptyset; R_R \leftarrow \emptyset;
                   end if
                                                                                                                                             > Compute and add evaluation rules for main operator
14:
                   Cells \leftarrow op's-evaluation-tables
 15:
                  for all cell \in Cells do
 16:
                          Convert cell to single rule r_e, substituting formula names
 17:
                           R_E \leftarrow R_E \cup r_e
 18:
                   end for
 19:
                   if \phi-is-main-formula then
20:
                          \begin{array}{c} R_E \leftarrow R_E \cup ([\phi]T \rightarrow SUCCESS) \\ R_E \leftarrow R_E \cup ([\phi]F \rightarrow FAILURE) \\ P \end{array}
20:
21:
22:
                          R_E \leftarrow R_E \cup ([\phi]? \rightarrow REPEAT)
23:
                   end if
                                                                                                                                                                      > Compute initial state for this subsystem
24:
25:
26:
27:
                   \text{ if } op = a \text{ then } S \leftarrow R[a] 
                  else if op = !a then S \leftarrow R[!a]
                 else if op \in \{\vee, \wedge\} then S \leftarrow S^1 \cup S^2 \cup R[\phi]B
else if op \in \{\vee, \wedge\} then S \leftarrow S^1 \cup S^2 \cup R[\phi]A
28:
29:
                 else if op \in \{\Box, \Diamond\} then S \leftarrow S^1 \cup R[\phi]
else if op \in \{X, W\} then S \leftarrow R[\phi]
30:
                   end if
                                                                                                                                         Compute and add reactivation rules for main operator
                  \begin{array}{l} \text{if } op \in \{\vee, \wedge\} \text{ then } R_R \leftarrow R_R \cup ([\phi]?Z \rightarrow R[\phi]?Z), \text{ for } Z \in L, R, B \\ \text{else if } op = U \text{ then } R_R \leftarrow R_R \cup ([\phi]?Z \rightarrow R[\phi]?Z, S^1, S^2), \text{ for } Z \in A, B, L, R \\ \text{else if } op \in \{\Box, \Diamond\} \text{ then } R_R \leftarrow R_R \cup ([\phi]? \rightarrow R[\phi], S^1) \\ \text{else if } op \in \{X, W\} \text{ then } R_R \leftarrow R_R \cup ([\phi]? \rightarrow R[\phi]M, S^1) \cup ([\phi]?M \rightarrow R[\phi]M) \\ \end{array} 
31:
32:
33:
34:
35:
                  end if
                                                                                                                                                                                                    ▷ Return computed system
return \langle R_E, R_R, S \rangle
```

RuleRunner generates several rules for each operator, but this number is constant, as it corresponds to the size of evaluation tables plus special rules (like the SUCCESS one). The number of rules corresponding to $\phi \lor \psi$, for instance, does not depend in any way on the nature of ϕ or ψ , as only the final truth evaluation of the two subformulae is taken into account. The preprocessing phase creates the parse tree of the property to encode and adds a constant number of rules for each node (subformula). The obtained rule set does not change at runtime nor when monitoring new traces. During the runtime verification, for each cell of the trace the system goes through all rules exactly once. This is guaranteed by the post-order visit of the parsing tree, as shown in Algorithm 2, assuring pre-emption for rules evaluating simpler formulae. Therefore, the complexity of the system is inherently linear. This is not in contrast with known exponential lower bounds

for the temporal logic validity problem, as RuleRunner deals with the satisfiability of a property on a trace, thus tackling a different problem from the validity one (this distinction is also mentioned in [9]). A prototype for RuleRunner can be found here.

As an example, consider the formula $\phi = a \lor \Diamond b$ and the trace t = [c - a - a]b, d-b, END (dashes separate cells and commas separate observations in the same cell). If monitoring ϕ over t, a fails in the first cell, while b is sought until the third cell, when it is observed. Thus the monitoring yields a success even before the end of the trace.

EVALUATION RULES

- $R[a], a \text{ is not observed} \rightarrow [a]F$
- $R[b], b \text{ is observed} \rightarrow [b]T$
- $-R[b], b \text{ is not observed} \rightarrow [b]F$
- $\begin{array}{l} -R[\Diamond b], \ b]T \to [\Diamond b]T \\ -R[\Diamond b], \ b]F \to [\Diamond b]? \end{array}$
- $\begin{array}{l} -R[a\vee\Diamond b]R, [a]F, [\diamond b]? \rightarrow [a\vee\Diamond b]?_R \\ -R[a\vee\Diamond b]R, [\diamond b]T \rightarrow [a\vee\Diamond b]T \\ -R[a\vee\Diamond b]R, [\diamond b]? \rightarrow [a\vee\Diamond b]?_R \end{array}$
- $[a \lor \Diamond b]T \to SUCCESS$

REACTIVATION RULES

$$\begin{split} [\Diamond b]? &\to R[b], R[\Diamond b] \\ [a \lor \Diamond b]?_R &\to R[a \lor \Diamond b]R \end{split}$$

INITIAL STATE

- $R[a], R[b], R[\Diamond b], R[a \lor \Diamond b]B$

EVOLUTION OVER [c - a - b, d - b, END]

state	$R[a], R[b], R[\Diamond b], R[a \lor \Diamond b]B$
+ obs	$[R[a], R[b], R[\Diamond b], R[a \lor \Diamond b]B, c$
eval	$[a]F, [b]F, [\Diamond b]?, [a \lor \Diamond b]?_R$
react	$R[b], R[\Diamond b], R[a \lor \Diamond b]R$
state	$[R[b], R[\Diamond b], R[a \lor \Diamond b]R$
+ obs	$R[b], R[\Diamond b], R[a \lor \Diamond b]R, a$
eval	$[b]F, [\Diamond b]?, [a \lor \Diamond b]?_R$
react	$R[b], R[\Diamond b], R[a \lor \Diamond b]R$
state	$R[b], R[\Diamond b], R[a \lor \Diamond b]R$
+ obs	$R[b], R[\Diamond b], R[a \lor \Diamond b]R, b, d$
eval	$[b]T, [\Diamond b]T, [a \lor \Diamond b]T, SUCCESS$
STOP	PROPERTY SATISFIED

The behaviour of the runtime monitor is the following:

- At the beginning, the system monitors $a, b, \Diamond b$ and $a \lor \Diamond b$ (initial state = $R[a], R[b], R[\Diamond b],$ $R[a \lor \Diamond b]B$). The -B in $R[a \lor \Diamond b]B$ means that both disjuncts are being monitored.
- In the first cell, c is observed and added to the state S. Using the evaluation rules, new truth values are computed: a is false, b is false, $\Diamond b$ is undecided. The global formula is undecided, but since the trace continues the monitoring goes on. The -Rin $R[a \lor \Diamond b]R$ means that only the right disjunct is monitored: the system dropped a, since it could only be satisfied in the first cell.
- In the second cell, a is observed but ignored (the rules for its monitoring are not _ activated); since b is false again, $\Diamond b$ and $a \lor \Diamond b$ are still undecided.
- In the third cell, d is ignored but observing b satisfies, in cascade, b, $\Diamond b$ and $a \lor \Diamond b$. The monitoring stops, signalling a success. The rest of the trace is ignored.

4 **Encoding the monitor in a Neural Network**

The translation of a RuleRunner monitor in an equivalent neural network is composed by two main steps, respectively encoding a rule system in a logic program and the resulting logic program in a neural network. We hereby describe the details of the first step and explain how to use CILP for the second one. The first step of the neural encoding is the translation of a RuleRunner system into an equivalent logic program (Algorithm 3).

8

Algorithm 3 From RuleRunner to Logic Programs

1: function $RR2LP(\phi)$ 2 Create $RR_{\phi} = \langle R_E, R_R, S \rangle$ encoding ϕ 3: Create an empty logic program LP $\triangleright C_E$ for all $R[\phi], [\psi^1]V, \dots, [\psi^n]V, \rightarrow [\phi]V \in R_E$ do $LP \leftarrow LP \cup [\phi]V :\sim UPDATE], R[\phi], [\psi^1]V, \dots, [\psi^n]V$ 4: 5: 6: end for $\triangleright C_P$ 7: for all $o \in R_E \cup R_R$ do $LP \leftarrow LP \cup o := \sim [U], o$ 8: end for 9: 10: for all $R[\phi] \in R_E \cup R_R$ do $LP \leftarrow LP \cup R[\phi] := \sim [UPDATE], R[\phi]$ 11: 12: end for $\triangleright C_R$ $\begin{array}{l} \text{for all } [\phi]? \rightarrow R[\phi], R[\psi^1], \dots, R[\psi^n] \in R_R \text{ do} \\ LP \leftarrow xLP \cup R[\phi] : [UPDATE], [\phi]? \end{array}$ 13. 14: 15: for all $R[\psi^i]$ do 16: $LP \leftarrow LP \cup R[\psi^i] := [UPDATE], [\phi]?$ 17: end for 18: end for 19: $LP_{\phi} = LP$ 20: return LP_{ϕ} 21: 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 3). 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 $x :- \sim [UPDATE], x$, where x is is a rule name $(R[\phi])$ or an observation ($o \in O$). 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 memory 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.

The example in the previous section corresponds to the logic program $LP_{(a \lor \Diamond b)}$, depicted below ([UPDATE] shortened to [U]).

EVALUATION CLAUSES

[a]F	$:- \sim [U],$	$R[a], \sim a$	

- $\begin{array}{l} [a]F':\sim\sim [U], R[a],\sim a\\ [b]T:\sim [U], R[b], b\\ [b]F:\sim [U], R[b], \sim b\\ [\langle \phi b]T:\sim [U], R[\langle \phi b], [b]T\\ [\langle \phi b]?:\sim [U], R[\langle \phi b], [b]F\\ [a \lor \langle \phi b]?_R:\sim [U], R[a \lor \langle \phi b]R, [a]F, [\langle \phi b]?\\ [a \lor \langle \phi b]T:\sim [U], R[a \lor \langle \phi b]R, [\langle \phi b]T\\ [a \lor \langle \phi b]?_R:\sim [U], R[a \lor \langle \phi b]R, [\langle \phi b]?\\ SUCCESS:\sim [U], [a \lor \langle \phi b]T \end{array}$
- _

REACTIVATION CLAUSES

```
\begin{array}{ll} &-& R[\Diamond b]: \text{-} [U], [\Diamond b]?\\ &-& R[b]: \text{-} [U], [\Diamond b]?\\ &-& R[a \lor \Diamond b]R: \text{-} [U], [a \lor \Diamond b]?_R \end{array}
```

```
PERSISTENCE CLAUSES
```

- $\begin{array}{l} \ a: \sim \sim [U], a \\ \ b: \sim \sim [U], b \\ \ R[a]: \sim \sim [U], R[a] \\ \ R[b]: \sim \sim [U], R[b] \\ \ R[\diamond]: \sim \sim [U], R[\diamond b] \\ \ R[\diamond \diamond b] B: \sim \sim [U], R[\diamond b] \\ \ R[a \lor \diamond b] B: \sim \sim [U], R[a \lor \diamond b] B \\ \ R[a \lor \diamond b] R: \sim \sim [U], R[a \lor \diamond b] R \end{array}$

Summarising, we start from a formal property ϕ expressed as an FLTL 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} . The correctness of this translation can be proved by induction on the monitoring process; we omit such proof due to lack of space. We can then exploit the CILP algorithm [8] to translate the logic program LP_{ϕ} into an equivalent neural network NN_{ϕ} .

Algorithm 4 CILP System - Translation

```
1: function CIL^2P(LP_{\phi})
2:
3:
        Create empty network NN
        for all clause c \in LP_{\phi}, c = h := b_1, \ldots, b_n do
4:
            Add a neuron C to the hidden layer
5:
            If not there, add a neuron H to the output layer
6:
7:
8:
            Connect C to H
            for all clause b_i \in c do
               If not there, add a neuron B_i to the input layer
9:
               Connect B_i to C
10:
             end for
11:
         end for
12:
         NN_{\phi} = NN
13: return NN_{\phi}
14: end function
```

Each clause (c) of LP_{ϕ} is mapped from the input layer to the output layer of NN_{ϕ} through one neuron (labelled C) in the single hidden layer of NN_{ϕ} . We omitted the computation of the parameters; intuitively, the Translation Algorithm from LP_{ϕ} to NN_{ϕ} has to implement the following conditions: (C1) the input potential of a hidden neuron (C) can only exceed C's threshold, activating C, when all the positive antecedents of c are assigned the truth value true while all the negative antecedents of c are assigned false; and (C2) the input potential of an output neuron H can only exceed H's threshold, activating H, when at least one hidden neuron C that is connected to H is activated; the translation algorithm is sound and complete [8].

 $LP_{(a \lor \Diamond b)}$ is translated into the neural network $NN_{(a \lor \Diamond b)}$ in Fig. 4.

The result of the neural encoding phase is a neural network NN_{ϕ} able to perform runtime verification, monitoring ϕ over the given trace(s). The general algorithm for

10

neural monitoring is described in Algorithm 5. Adding x to a given layer means activating the neuron corresponding to x in that layer. In terms logic programming, this corresponds 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 alternation 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. Moving to the next trace cell is achieved by firing all reactivation rules in parallel once.



Fig. 4: (simplified) Neural network for monitoring $a \lor \Diamond b$: $NN_{(a\lor \Diamond b)}$. The input and output layers include neurons whose labels correspond to atoms in the logic program; active neurons are filled in grey, representing the initial state

 $R[a], R[b], R[\Diamond b], R[a \lor \Diamond b]. [U]$ stands for [UPDATE] and [S] for [SUCCESS]. Hidden neurons correspond to clauses, labelled with e (resp. p,r) to mark that it encodes an evaluation (resp. persistence, reactivation) clause. Solid lines represent connection with positive weights, dashed lines represent negative weights. To simplify the

visualisation, all recurrent connections and connections from [U] are omitted in Fig. 4. For instance, the leftmost hidden neuron in Fig. 4 has ingoing connections from a

(negative weight) and R[a] (positive weight), plus the negative connection from [U](omitted in the figure); it has an outgoing connection towards [a]F: this corresponds to the clause $[a]F :- \sim [UPDATE], R[a], \sim a$, the first evaluation clause in $LP_{(a \lor \Diamond b)}$.

Algorithm 5 Runtime Verification using NN_{ϕ}

1:	function NN-MONITOR(ϕ ,trace t)
2:	Create $RR_{\phi} = \langle R_R, R_E, S \rangle$ encoding ϕ (Algorithm 2)
3:	Rewrite RR_{ϕ} into LP_{ϕ} (Algorithm 3)
4:	Rewrite LP_{ϕ} into NP_{ϕ} (Algorithm 4: $CILP$)
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

5 Experiments

5.1 Parallel Monitoring

The encoding of RuleRunner in a neural network allows for the monitoring of several traces in parallel. A cell in a trace is a vector of observations which is fed to the neural network. To monitor multiple traces, it is sufficient to compose all observation vectors (of different traces at the same time) as rows in a matrix. The monitoring is then carried out as explained above, with all vector-matrix steps substituted by matrix-matrix operations. Furthermore, the matrix-based nature of neural networks allows for straightforward optimisations in order to speed up the monitoring: we focused on sparse form and GPU computation, using Matlab 2013a for all experiments. A sparse matrix is a matrix primarily populated by zeros (e.g. any identity matrix). The *sparse(m)* function is a lossless compression that stores only the non-null elements in a matrix m. The weight matrices in our networks are sparse: the size depends on the complexity of the encoded formula (and therefore the number of rules in the monitor), but the number of non-null elements per row/column is constant, as each rule (clause) only involves truth values of direct subformulae and has a constant number of literals in the body/head. We also exploited MathWork's Parallel Computing Toolbox to run our code on GPU. We used GPUArray, a special array type with several associated functions that allows to perform computations on CUDA-enabled NVIDIA GPUs directly from MATLAB. In the following set of experiments, the *base* implementation for ϕ is the neural-network encoding a monitor for ϕ as described in the previous section (NN_{ϕ}) ; the sparse and *GPU* implementations were simply obtained, respectively, the $sparse(\cdot)$ and $qpuArray(\cdot)$ functions to all the matrices in the base implementation.



Fig. 5: Scalability experiments for matrix-based monitors

In terms of code, the difference between the three implementations is deliberately minimal, as our goal is to show how tiny modifications to the code allow for performance improvement. For this experiments, we generated random formulae from the FLTL grammar and sets of observations: rather than testing domain-specific properties, we aimed at checking a set of unbiased formulae. The complexity of the temporal formulae is measured in terms of *leaves* of the parsing tree: for instance, $\Box(a \lor b)$ has two leaves $(a, b), \Diamond (Xa \land ((c \lor d)Ud))$ has four leaves (a, b, c, d), et cetera. In Figure 5.A, we show how the length of the trace has no impact on the average time required to monitor a cell: the resources required by our system do not grow over time. In Figure 5.B, we measured how increasing the complexity of the formula impacts on the average time required to monitor a cell. The first striking feature of these curves is that, while for simple formulae *base* and *sparse* are significantly faster than *gpu*, for higher number of leaves *base* ends up having average times similar to those of *gpu*. Moreover, *base*'s curve looks steeper than the other two. The same values can be averaged over the number of leaves: that is, given the average time required to monitor a cell when monitoring a formula with n leaves, we can divide that time value by n. The chart in Figure 5.C shows how the gpu curve significantly lowers the time per cell per leaf, when the number of leaves increases. Our explanation for this behaviour is that the GPU implementation suffers from a remarkable overhead in the CPU-GPU communication: this delta becomes less influent when making the formula more complex and thus the matrices bigger. Finally, it is relevant to measure the increase rate of these curves: to do so, we divided all measured times by the first experiment in each curve (Figure 5.D). We observed that *base* scales linearly w.r.t. the number of leaves: for instance, the experiment with 32 leaves is 16 times slower than the experiment (with the same implementation) with 2 leaves. The curve for the *sparse* form is more gradual, with the last experiment being ~ 4 times slower than the first one. But the implementation with the smoothest curve is the gpu-based one, with the 32-leaves experiment being only ~ 1.5 times slower than the 2-leaves one.

We ran other tests to study the impact of other features, such as the number of consecutive *next* operators or the interleaving of temporal and classical operators. All tests generated curves following the same patterns. We thus observed how the *sparse* form allows for a speed-up of the monitoring, while the *gpu* implementation shows the best scaling factor.

5.2 Learning as property adaptation

Neural networks also offer intrinsic learning capabilities. In our system, observations (events) are presented to the monitor as the activation of labelled input neurons. The feedforward propagation then causes some output neurons to be activated, representing the result of the monitoring of the encoded formula on the given cell. It may be the case that a user or domain expert would like to modify the verdict of a monitoring task: for instance, consider a security system built to detect consecutive *login* operations from the same user with no *logout* in between: $\phi = login \Rightarrow X(!login \cup logout)$. Now suppose the system detects a number of violations, in a given LAN, that the security manager judges as false positives. For instance, that LAN could be a lab where users log in from different devices, or it could be a trustable section of the network. It is reasonable that the security manager wants to maintain the security system, but also to relax the formal property in order to include logins from the given LAN, and therefore avoid false positives and improve efficiency. Reclassifying a trace means demoting the causality relation between the observations and the actual output while promoting the relation between observations and expected output: it is, in fact, a supervised learning task.



Fig. 6: Simplified network for the *consecutive logins* example, representing the current login (*in*), logout (*out*), the existence of a previous login (*prv*), the occurrence of a login without a logout before (*v*1, corresponding to a violation of $!login \cup logout$), the occurrence of v1 when a previous login occurred (*v*2, corresponding to $!\phi$) and, finally, flags for global violation ([*F*] for *failure*) and success ([*S*]).

The security manager tells the system to accept the consecutive logins occurring in LAN. A new input neuron, *lan* is added to the system. The learning task is performed as training of two perceptrons: P1 maps $\{e1, e2, e3, e4\}$ onto [F], while P2 maps the same hidden neurons onto [S]. The perceptrons are conceptually isolated by the rest of the network as follows: the output is the flag to be modified, and the inputs are all hidden neurons in the network. Note that, even if all hidden neurons are used as inputs for a perceptron and its learning algorithm, only the weights of the connections from active neurons can be modified, thus limiting the algorithm to the relevant (currently active) hidden neurons. P1 is trained to accept only traces where e_3 , $!e_4$ occurs, while for P2 the positive examples involve e_{3}, e_{4} . Both perceptrons are trained with the standard perceptron learning rule: Figure 6 shows the results when applying the learning rule with $\eta = 0.2$ and thresholds = 0.5. Relevant connections are labeled with their weights; connections with weight close to 0 are omitted. For P1, the weights from e_{1}, e_{2}, e_{3} are only slightly modified by the learning; the connection from e4, on the contrary, goes from being 0 to a negative value. For P2, since the network is trained to associate the co-occurence of the violation $v2 = !\phi$ and *lan*, the weights from e3 and e4 towards [S] are increased. As a result, when *lan* is not active the network displays the nominal behaviour and signals a violation ([F]); on the contrary, when both lan and v2 are active, the negative weight from e4 prevents [F] from being activated, while the sum of the signals from e3 and e4 activates [S].

6 Conclusions and Future Work

Firstly, we developed a runtime monitoring system, RuleRunner, to monitor FLTL properties on finite traces. Secondly, we discussed the encoding, partially based on CILP's Translation Algorithm, of a RuleRunner rule system in a standard feedforward neural network with recurrent connections and one hidden layer. Thirdly, we showed how matrix-based optimisations allows for major improvements in terms of performance, either concerning actual speed (matrices in sparse form) or scaling factor (GPU computation). Finally, we pointed out how in our framework learning corresponds to property adaptation. A first direction for future work is to adopt and compare more advanced techniques for

parallel monitoring, such as the MapReduce framework (recently applied to Runtime Verification in [1]) and the parallel algorithms in [5]. A second direction for future work is the development of ad-hoc learning strategies, mostly based on the Backpropagation algorithm [11], for specific classes of adaptation problems, such as soft norms in BPM [19] and false positives in Security [6].

References

- B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, and S. Hall. Mapreduce for parallel trace validation of ltl properties. In *Procs. of Runtime Verification 2012*, pages 184–198.
- 2. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *Journal of Logic and Computation*, volume 20:pages 675–706, 2010.
- A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Procs. of Runtime Verification 2007*, pages 126–138.
- A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In Procs. of Foundations of Software Technology and Theoretical Computer Science 2006, pages 260–272.
- S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Gpu-based runtime verification. In Procs. of the IEEE International Symposium on Parallel and Distributed Processing 2013, pages 1025–1036.
- D. Breitgand, M. Goldstein, and E. H. Shehory. Efficient control of false negative and false positive errors with separate adaptive thresholds. *Network and Service Management, IEEE Transactions on*, 8:128–140, 2011.
- A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng. Deep learning with cots hpc systems. In *Procs. of International Conference on Machine Learning 2013*, pages 1337–1345.
- A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, volume 11:pages 59–77, 1999.
- 9. D. Drusinsky. The temporal rover and the atg rover. In *Procs. of the International Workshop* on SPIN Model Checking and Software Verification 2000, pages 323–330.
- C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Procs. of Computer-Aided Verification 2003*, pages 27–39.
- 11. S. Haykin. Neural Networks: A Comprehensive Foundation (3rd Edition). 2007.
- P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. Dynamite: Dynamic task nets for software process management. In *Procs of International Conference on Software Engineering* 1996, pages 331–341.
- Q. V. Le, M. Ranzato, R. Monga, M. Devin, G. Corrado, K. Chen, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning. In *Procs. of International Conference on Machine Learning 2012.*
- 14. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, volume 78:pages 293–303, 2009.
- O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *in Procs. of Logic of Programs 1985*, pages 196–218.
- 16. J. Lukasiewicz. O logice trjwartosciowej (On Three-Valued Logic). 1920.
- 17. A. Pnueli. The temporal logic of programs. In *Procs. of the Annual Symposium on Foundations* of Computer Science 1977, pages 46–57.
- G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, volume 70:pages 119–165, 1994.
- W. M. P. van der Aalst et.al. Process mining manifesto. In Procs of Business Process Management Workshops 2011, pages 169–194.