A Framework for Monitoring and Learning in Multiagent Systems

Alan Perotti[1], Guido Boella[1], and Artur d'Avila Garcez[2]

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

Abstract. Open Multiagent Systems are systems designed with a general purpose in mind but with an unknown population of autonomous agents at design time. According to specific implementation and nature, each agent finds a way to implement the assigned task. Autonomy has to be counterbalanced by mechanisms for control and regulation of the agents: Normative Multiagent Systems include techniques for enforcing norms and sanctioning agents in case of violations. We claim that these approaches do not fully capture nor exploit the richness of behaviours that arise from an open, heterogeneous multiagent system: in our vision, norms should be adaptive, and they should be inferred taking into account the behaviour of agents showing better performances. In this paper, we propose an abstract framework where the agents are monitored in real-time and the norms can be refined by observing the agents with best fitness. We also present an implementation of our framework using Linear Temporal Logic, Runtime Verification and Neural-Symbolic Integration in an Iterative Development cycle.

1 Introduction

From the perspective of intelligent agents, autonomy is a key factor in the resolution of complex problems and it allows a flexible behaviour in a number of theoretical scenarios and practical application areas [32]. From the point of view of the designer of a multi-agent system (MAS), autonomy has to be counterbalanced by mechanisms for control and regulation of the agents. In particular, one prominent challenge is how to configure and organise MAS, given their constantly changing structure [18]. One leading solution is to employ the use of norms. In human societies, norms are essential to regulation, coordination, and cooperation; normative multi-agent systems (NMAS, [18]) combine norms and MAS. In NMAS, the regimenting approach [22] is based on strict monitoring and enforcement of norms by means of sanctioning techniques. However, in real life it is hardly the case that a normative code accounts for the perfect and optimal behaviour: in general, norms are not ideal and leave room for improvement and refinement. Approaches based on soft norms [5] implement a more tolerant regulation, including peculiarities as contrary-to-duty, exceptions, and the deontic obligation/permission dualism. Therefore, soft norms are flexible and dynamic, but not ideal, and adaptation has to be guided by an external source, such as an oracle or a domain expert. When dealing with a MAS, the violation of norms, and -more generally- the display of a non-scripted behaviour, is a natural consequence of the agents being autonomous. Furthermore, many agents are built to optimise a task given a set of requirements [32]. It is worth stressing that there are several reasons that may cause unpredictability in the behaviour of an agent; for instance, in the area of planning, possible causes include:

- **Bona-fide deliberation**, e.g. if the agent is given a plan based on Disjunctive Temporal Network [8] or over-subscription [10].
- **External causes**: plan failures due to dynamic or non-deterministic environment [29].
- **Malicious behaviour**, for agents in a competitive environment, when the intentional violation of norms is used to harm the competing agent(s) [25].

In the regulative approach, agents that do not comply with norms are sanctioned (or have to deal with additional norms), while well-behaving agents are valued according to their performance outcomes [19]. We claim that this approach does not fully capture nor exploit the richness of behaviours that arise from an open, heterogeneous MAS. In our vision, adaptive norms and autonomous agents can be integrated: a NMAS can exploit the unscripted behaviour of agents and use the best outcomes as a benchmark to adapt the regulation over the whole MAS. This is a common approach in real human societies, from huge institutions (for instance, the subsidiarity principle in the European Union Law) to private companies. For example, in a bank, the MIFID[1] 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 bank branch considered successful and compliant to hard constraints, substituted scanned versions of a given documents to paper ones despite the original specifications, such specifications may be revised. This adaptation integrates the original normative code with the the traces of the activity of the successful branch.

Our Research Question is the following:

How can we exploit the intelligent behaviour of agents, learning from the best-performing agents in an open MAS?

This breaks down to the following sub-questions:

1. How can we monitor, in real time, the performance of several heterogeneous agents executing actions in parallel?
2. How can we learn a 'good violation' when an agent achieves a good outcome in an unpredicted way, and how can we integrate it with the designer's high-level knowledge?

[1] Markets in Financial Instruments Directive

Methodology: In this paper, we introduce a framework for monitoring and learning by adaptation in a MAS setting. We rely on Runtime Verification [20] for the monitoring module, as it has to be based on the observation of traces (of agents' actions). We adopt Temporal Logic [24] as the meta-knowledge given to the agents as a guide in their local planning process; we do so since time-based reasoning is essential in all planning frameworks [11]. We use the Neural-Symbolic paradigm [14] to encode a Runtime Verification monitor in a standard neural network. The advantage of using neural networks is twofold: parallel verification and off-the-shelf learning strategies.

Scope: We are interested in putting together agent autonomy, iterative design and learning by imitation. Our framework applies to a number of areas: business process management, fraud detection, user modelling, etc.. In this paper, we present a more detailed scenario based on planning and run-time monitoring, implemented with neural networks. We focus on the designer's perspective and the coordination level: issues concerning agent-level tasks, such as the computation of the plan, are outside the scope of this paper. Furthermore, we will not address the issue of outcome evaluation: we are interested in providing a framework where emergent behaviours are learned, and deciding which plan should be labelled *successful* is a domain-dependent issue.

The remainder of the paper is organised as follows: Section 2 provides an overview of our framework, while Section 3 presents the involved methodologies. Section 4 gives a technical introduction of our runtime verification system, RuleRunner, detailing neural encoding and monitoring, and introducing learning. Section 5 ends the paper with a final discussion.

2 Framework

We propose a framework for monitoring and learning by adaptation in a MAS setting. The abstract interaction loop is visualized in Figure 1.

The leftmost block is the MAS-designer (D), the rightmost one is a simple agent (A_i). According to different scenarios, the designer could be a human in the loop (e.g., the well-known 'domain expert') or an agent with a special role. The frameworks depicts a single agent, but the designer is meant to carry on many parallel, one-to-one interaction loops with all agents in a MAS (that is, assume that $A_i \in A_1, .., A_n$).
In the beginning, D establishes some abstract information ϕ about the task to perform (e.g. a normative code, a partial plan, a set of constraints). D then computes a monitor for ϕ and sends it to all agents. Each agent A_i, according to her internal structure, uses the monitor to build a plan to implement ϕ . We assume that the internal structure of A_i is not known nor observable: from our perspective, A_i is a black-box who is given ϕ and that subsequently interacts

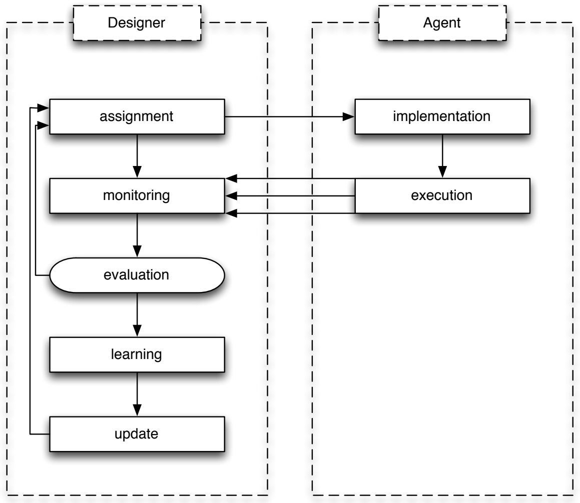

Fig. 1: Abstract interaction framework

with the environment, executing a plan she computed. The only way D has to evaluate A_i's behaviour is to observe her actions, compare them with ϕ, and collect information about the final outcome OUT_i of A_i's performance: e.g. the resources used to execute a plan, the final price paid in an auction, winning a debate, etc. OUT_i is considered *good* or *bad* by comparison with some benchmark OUT or some objective function to be optimised. At this point (evaluation), for each agent A_i, D knows whether A_i violated ϕ and how good A_i's outcome is. If there were no violations then A_i's behaviour was nominal: nothing unexpected, no major failures, no bad behaviour. In case of violations, if the outcome is bad, then A_i made some sub-optimal decisions, violating ϕ and obtaining a bad outcome. In these three cases, no useful feedback is provided and the loop starts again. If ϕ is violated and OUT is outperformed, then A_i found a detour from ϕ that paid off: in this case, D should learn what A_i's improvements are (ϕ_i), and use them as future benchmark for the next iterations, sending the adapted monitoring module to all agents in the MAS.
Zooming out, our frameworks allows an open-MAS to implement a task in an autonomous way starting from some abstract task description ϕ, while the designer D monitors the execution of all plans in parallel and, if some deviation ϕ_i from the benchmark leads to an improvement in performance ($OUT_i > OUT$), it tries to learn it and makes it the starting point for the following iterations ($\phi ::= \phi_i$).

3 Involved Methodologies

Our framework shares many concepts with specific research areas:

1. **Plan recognition** [7],[33] identifies a number of contexts in which an agent's plan has to be inferred from the mere observation of her actions: we take this to a higher level, as we infer some meta-level temporal knowledge about the executed actions.
2. **Continual planning** [6],[30] embeds planning in a loop involving sensing and belief update: we transpose this concept to the designer's perspective.
3. **Software engineering** developed the iterative design [28] cycle, linking reasoning and learning in a loop.
4. **Temporal meta-knowledge** is shown to be a valuable starting point for planning [1]. Furthermore, in [1] Bacchus and Kabanza claim that "an important area for future research will be to employ learning and reasoning techniques to automatically generate this domain-specific knowledge".
5. **Genetic algorithms** [15], mimicking natural selection, aim to optimise the solution to a problem by letting a population of candidate solutions mutate randomly, then using the fittest individuals in the following generation - loop.

Concerning the technical side, we adopt techniques from (rule-based) runtime verification, neural networks, and neural-symbolic integration.

3.1 Rule-based Verification

Like Model Checking, Runtime Verification (RV) relates an observed system with a formal property ϕ (often formulated in linear temporal logic LTL [24]), determining the semantics of ϕ while executing the system under scrutiny. The black-box, real-time nature of RV poses additional constraints w.r.t. the general Model Checking area: since the inner structure of the observed system is unknown, it is not possible to build an automaton for it, nor to 'peek in the future' and use path exploration techniques. An RV module, or monitor, is defined as a device that reads a finite trace and yields a certain verdict [20]. 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. 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 [21], RVLTL [4], LTL3 [3], LTL± [12] just to name some. Since 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 ($X, \bar{X}$ in [4], X, W 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 $u = a_0..a_{n-1}$ denote a finite trace of length n. The truth value of an FLTL formula ψ (either $X\phi$ or $W\phi$) wrt. u at position $i < n$, denoted by $[u, i \models \psi]$, is an element of $\mathbb{B}$ and is defined as follows:

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

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

In particular, since FLTL extends LTL, every LTL formula is a FLTL formula, while $FLTL \setminus LTL$ is the the set of formulae where W occurs.

3.2 Neural networks

An artificial Neural Network (NN, [16]) 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 novel 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.
In our framework, neural networks play a role in several tasks: they allow to monitor several agents in parallel and offer off-the-shelf learning strategies,; furthermore, when the Designer sends the updated monitor to all agents, it is sufficient to send the new weight matrix of the network encoding the monitor.

3.3 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 [27], 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 [9]. CILP builds upon KBANN and [17] to provide a sound theoretical foundation for reasoning in artificial neural networks and learning capabilities.

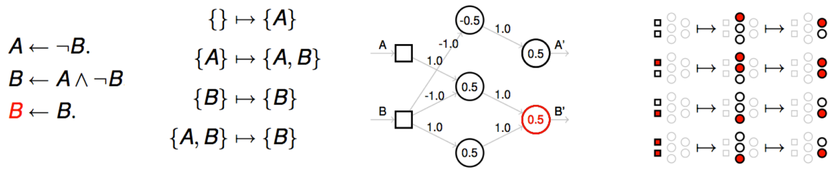

Fig. 2: Logic program as a network (CILP)

In particular, rules are mapped onto hidden neurons, the preconditions of rules onto input neurons and the conclusion of the rules onto output neurons (example in Figure 2). The weights are then adjusted to express the dependence among all these elements.

CILP provides an algorithm to translate a propositional logic program in a neural network that computes the stable model of the initial program. 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.

4 Reasoning and Learning

4.1 Runtime Verification using a Rule System

The first phase of our approach consisted in the design and development of a rule-based runtime monitor, RuleRunner, suitable to be encoded in a neural network. RuleRunner observes finite but expanding traces and returns an FLTL verdict. As it scans the trace, RuleRunner maintains a state composed by rule names, 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 S composed by 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 $[\psi]$ currently has the truth value V. The third truth value, ?, means that it is impossible to give a binary verdict in the current cell: for instance, if the formula is Xa, a has to be observed in the second cell, and it's impossible to evaluate Xa in the first cell.
The state evolves according to rules: RuleRunner is composed by evaluation and reactivation rules (R_E, R_R). Evaluation rules follow the pattern

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

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 follow the pattern

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

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. Thus a RuleRunner system is defined as a tuple $\langle R_E, R_R, S\rangle$ (state, evaluation and reactivation rules respectively). Given a (F)LTL formula ϕ and a trace t, the following algorithms summarise the creation and runtime behaviour of a RuleRunner system monitoring ϕ over t:

Algorithm 1 Construction of a RuleRunner system RR_ϕ

```
1: function RR-BUILD($\phi$)
2:     Parse the LTL formula $\phi$ in a tree
3:     Generate $R_E$, $R_R$ and the initial state $S$
4:     return $RR_\phi = \langle R_E, R_R, S\rangle$
5: end function
```

Algorithm 2 Runtime verification using RR_ϕ

```
1: function R-MONITOR($RR_\phi$,trace t)
2:     while new observations exist in t do
3:         Add observations to state
4:         Compute truth values using evaluation rules
5:         if state contains SUCCESS or FAILURE then
6:             return SUCCESS or FAILURE respectively
7:         end if
8:         Compute next state using reactivation rules
9:     end while
10: end function
```

As an example, consider the formula $\phi = a \vee \Diamond b$ and the trace $t = [c{-}a{-}b, d{-}b]$. Intuitively, ϕ means *either a now or b sometimes in the future.* If monitoring ϕ over t, a fails straight from the beginning, 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.

In RuleRunner, the formula ϕ is parsed into a tree, with $\vee$ as root and a, b as leaves. Then, starting from the leaves, evaluation and reactivation rules for each node are added to the (initially empty) rule system. In our example, (part of) the rule system obtained from ϕ, namely $RR_{(a \vee \Diamond b)}$, and its behaviour over t are the following:

EVALUATION RULES

- $R[a]$, a is not observed $\rightarrow [a]F$
- $R[b]$, b is observed $\rightarrow [b]T$
- $R[b]$, b is not observed $\rightarrow [b]F$
- $R[\Diamond b]$, $[b]T \rightarrow [\Diamond b]T$
- $R[\Diamond b]$, $[b]F \rightarrow [\Diamond b]?$
- $R[a \vee \Diamond b]B$, $[a]F$, $[\Diamond b]? \rightarrow [a \vee \Diamond b]?R$
- $R[a \vee \Diamond b]R$, $[\Diamond b]T \rightarrow [a \vee \Diamond b]T$
- $R[a \vee \Diamond b]R$, $[\Diamond b]? \rightarrow [a \vee \Diamond b]?R$
- $[a \vee \Diamond b]T \rightarrow SUCCESS$

REACTIVATION RULES

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

INITIAL STATE

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

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

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

The behaviour of the runtime monitor is the following:

- At the beginning, the system monitors a,b,$\Diamond b$ and $a \vee \Diamond b$ (initial state = $R[a], R[b], R[\Diamond b], R[a \vee \Diamond b]B$). The $-B$ in $R[a \vee \Diamond b]B$ 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 $-R$ in $R[a \vee \Diamond b]R$ 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 $a \vee \Diamond b$ are still undecided.
- In the third cell, d is ignored but observing b satisfies, in cascade, b, $\Diamond b$ and $a \vee \Diamond b$. The monitoring stops, signalling a success. The rest of the trace is ignored.

Most of the runtime verification approaches, like RuleR [2], are based on a (potentially exponential) tree-like structure of alternative hypotheses, while RuleRunner is rooted in maintaining a single state composed of the unique truth value of every subformula of the encoded property. Moreover, the *distributed* nature of a RuleRunner state and the local nature of rules (inferring the truth value of a formula from the truth values of its subformulae only) allows different rules to be applied in parallel. Moreover, the fixed number of elements (observations, rule names, truth evaluations) and the implicit representation of time (by means of rule reactivation) allow to build a network with fixed structure and features: this will be explained in the next subsections.

A technical report about RuleRunner, including formal results concerning semantics and complexity, is available on ArXiv [23]; an online tool also available[2].

4.2 Encoding a Rule System in a Neural Network

The second phase of our approach is to encode a RuleRunner system in a neural network. This translation is composed by two main steps, respectively encoding a rule system in a logic program and the resulting logic program in a neural network. In this section, we focus on the translation of a RuleRunner rule system in a logic program. CILP provides a sound and complete translation algorithm to encode a logic program into a (standard, feed-forward, recurrent) neural network, thus providing the second step in the general translation process.

The first step of the neural encoding is the translation of a RuleRunner system into an equivalent logic program (Algorithm 3).

[2] www.di.unito.it/∼perotti/RuleRunner.jnlp

Algorithm 3 From RuleRunner to Logic Programs

```
 1: function rr2lp($\phi$)
 2:     Create $RR_\phi = \langle R_E, R_R, S\rangle$ encoding $\phi$
 3:     Create an empty logic program $LP$
                                                                          ▷ $C_E$
 4:     for all $R[\phi], [\psi^1]V, \dots, [\psi^n]V, \rightarrow [\phi]V \in R_E$ do
 5:         $LP \leftarrow LP \cup [\phi]V$ :-$\sim[U], R[\phi], [\psi^1]V, \dots, [\psi^n]V$
 6:     end for
                                                                          ▷ $C_P$
 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
                                                                          ▷ $C_R$
13:     for all $[\phi]? \rightarrow R[\phi], R[\psi^1], \dots, R[\psi^n] \in R_R$ 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 for
19:     $LP_\phi = LP$
20: return $LP_\phi$
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 :- $\sim[UPDATE], x$ (shortened to *[U]* in the algorithm and in the next example).

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.

For instance, in the previous subsection we provided a simplified rule system for $a \vee \Diamond b$. Such system corresponds to the following logic program, $LP_{(a \vee \Diamond b)}$:

EVALUATION CLAUSES

- $[a]F$:- $\sim[U], R[a], \sim a$
- $[b]T$:- $\sim[U], R[b], b$
- $[b]F$:- $\sim[U], R[b], \sim b$
- $[\Diamond b]T$:- $\sim[U], R[\Diamond b], [b]T$
- $[\Diamond b]?$:- $\sim[U], R[\Diamond b], [b]F$
- $[a \vee \Diamond b]?R$:- $\sim[U], R[a \vee \Diamond b]B, [a]F, [\Diamond b]?$
- $[a \vee \Diamond b]T$:- $\sim[U], R[a \vee \Diamond b]R, [\Diamond b]T$
- $[a \vee \Diamond b]?R$:- $\sim[U], R[a \vee \Diamond b]R, [\Diamond b]?$
- $SUCCESS$:- $\sim[U], [a \vee \Diamond b]T$

REACTIVATION CLAUSES

- $R[\Diamond b]$:- $[U], [\Diamond b]?$
- $R[b]$:- $[U], [\Diamond b]?$
- $R[a \vee \Diamond b]R$:- $[U], [a \vee \Diamond b]?R$

PERSISTENCE CLAUSES

- a :- $\sim[U], a$
- b :- $\sim[U], b$
- $R[a]$:- $\sim[U], R[a]$
- $R[b]$:- $\sim[U], R[b]$
- $R[\Diamond b]$:- $\sim[U], R[\Diamond b]$
- $R[a \vee \Diamond b]B$:- $\sim[U], R[a \vee \Diamond b]B$
- $R[a \vee \Diamond b]R$:- $\sim[U], R[a \vee \Diamond b]R$

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. 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 [9] 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:     Create empty network $NN$
3:     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:         Connect $C$ to $H$
7:         for all clause $b_i \in c$ do
8:             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 [9].

In our working example, $LP_{(a \vee \Diamond b)}$ is translated into the neural network in Figure 3.

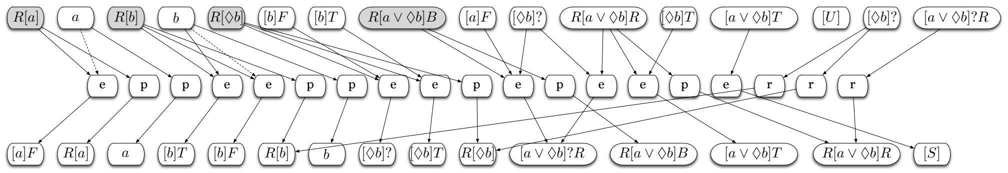

Fig. 3: (simplified) Neural network for monitoring $a \vee \Diamond b$: $NN_{(a \vee \Diamond b)}$

The input and output layers include neurons whose labels correspond to atoms in the logic program and the rule system: observations, rule names and truth evaluation. in Figure 3, active neurons are filled in grey, representing the initial state $R[a], R[b], R[\Diamond b], R[a \vee \Diamond b]$. $[U]$ stands for $[UPDATE]$ and $[S]$ for $[SUCCESS]$. Hidden neurons correspond to clauses: for the sake of explanation, each hidden neuron is 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 Figure 3. Recurrent links connect neurons in the output layer to the neuron in the input layer with the same label, while there is a positive connection (solid line) from $[U]$ to all r-labeled hidden neurons, and a negative connection (dashed line) from $[U]$ to all e and p-labeled hidden neurons (mirroring the $[UPDATE]$ switch in the logic program). For instance, the leftmost hidden neuron in Figure 3 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 $LP_{(a \vee \Diamond b)}$.

4.3 Runtime Verification using a Neural Network

The result of the neural encoding phase is a neural network NN_ϕ able to perform runtime verification. The general algorithm for 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 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. Moving to the next cell of the trace is achieved by firing all reactivation rules in parallel once.

Algorithm 5 Runtime Verification using NN_ϕ

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

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 analogous matrix-matrix operations. Furthermore, the matrix-based nature of a neural network allows for optimisations in order to speed up the monitoring. For instance, in RuleRunner, 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 4 shows the impact of increasing the size of the encoded formula on average time required to process 10000 cells of randomly-generated traces.

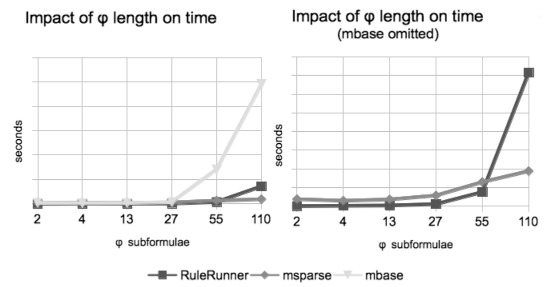

Fig. 4: Compared performances

4.4 Learning

The obtained system is a neural network encoding a RuleRunner monitor. The network has a standard architecture: one hidden layer, feedforward activation with recurrent connections from output to input layer. This allows to exploit a wide variety of learning strategies [16], most notably the backpropagation algorithm [26] and -due to the relevance of time in the monitoring task-, backpropagation though time [31,13]. In general, we are interested in the application of our framework to different scenarios, in order to test different setups of the learning component. For instance, if the encoded property is a set of norms, the learning goal would be to learn a soft violation while being as consistent as possible to the encoded norms. Another, completely different, scenario is learning the user interaction trajectories on a website: in this case the monitor encodes a set of interaction patterns, and if some user deviates from said patterns, the new trajectory is fed to the learning module. In this case it is not necessary to 'reinforce' the initial knowledge, and if enough users follow new patterns, the initially encoded ones can be discarded. Future work will be focused on the impact of the comparison of different learning strategies over specific domains.

5 Conclusions

We claim that multiagent systems should exploit the heterogeneous and autonomous nature of agents, monitoring in real-time the behaviour of all agents in a MAS and, when necessary, learning from the ones that perform better. In this paper, we propose an abstract framework that merges multiagent monitoring and property adaptation. We also present an implementation of our framework using Linear Temporal Logic, Runtime Verification and Neural-Symbolic Integration. We introduced a novel runtime verification system, RuleRunner, and showed how to encode it in a neural network, in order to exploit the parallel computation and learning strategies offered by the connectionist approach. Future lines of work focus on these themes: concerning the monitoring task, we are running experiments of GPU computation in order to improve performance; for the learning phase we are comparing off-the-shelf algorithms (most notably backpropagation and backpropagation through time) with ad-hoc learning strategies.

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191, Jan. 2000.
2. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.
3. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

4. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007.
5. G. Boella, G. Pigozzi, and L. van der Torre. Normative systems in computer science. ten guidelines for normative multiagent systems. In *Normative Multi-Agent Systems, Dagstuhl Seminar Proceedings 09121*, 2009.
6. M. Brenner and B. Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.
7. S. Carberry. Techniques for plan recognition. *User Model. User-Adapt. Interact.*, 11(1-2):31–48, 2001.
8. P. R. Conrad and B. C. Williams. Drake: An efficient executive for temporal plans with choice. *J. Artif. Intell. Res. (JAIR)*, 42:607–659, 2011.
9. A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Appl. Intell.*, 11(1):59–77, 1999.
10. M. B. Do and S. Kambhampati. Partial satisfaction (over-subscription) planning as heuristic search. In *Proceedings of KBCS-04*, 2004.
11. P. Doherty and J. Kvarnström. Temporal action logics. In *in Handbook of Knowledge Representation, Elsevier*, 2008.
12. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *CAV'03*, pages 27–39, 2003.
13. K. Fujarewicz and A. Galuszka. Generalized backpropagation through time for continuous time neural networks and discrete time measurements. In L. Rutkowski, J. H. Siekmann, R. Tadeusiewicz, and L. A. Zadeh, editors, *ICAISC*, volume 3070 of *Lecture Notes in Computer Science*, pages 190–196. Springer, 2004.
14. A. S. d. Garcez, D. M. Gabbay, and K. B. Broda. *Neural-Symbolic Learning System: Foundations and Applications.* Springer-Verlag New York, Inc., 2002.
15. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
16. S. Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007.
17. S. Hoelldobler and Y. Kalinke. Towards a new massively parallel computational model for logic programming. In *ECAI'94 workshop on Combining Symbolic and Connectioninst Processing*, pages 68–77, 1994.
18. C. D. Hollander and A. S. Wu. The current state of normative agent-based systems. *J. Artificial Societies and Social Simulation*, 14(2), 2011.
19. L. Laurian, J. Crawford, M. Day, P. Kouwenhoven, G. Mason, N. Ericksen, and L. Beattie. Evaluating the outcomes of plans: theory, practice, and methodology. *Environment and Planning B: Planning and Design*, 37(4):740–757, 2010.
20. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
21. O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In R. Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.
22. J. Olson and F. Svensson. Regimenting reasons. *Theoria*, 71(3):203–214, 2005.
23. A. Perotti. RuleRunner technical report. *ArXiv e-prints*, June 2013.
24. A. 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.
25. Z. Rui, T. Fu, D. Lai, and Y. Jiang. Cooperation among malicious agents: a general quantitative congestion game framework. In *Proceedings of the 11th International*

Conference on Autonomous Agents and Multiagent Systems - Volume 3, AAMAS '12, pages 1331–1332, 2012.

26. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.
27. G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artif. Intell.*, 70(1-2):119–165, 1994.
28. B.-Y. Tsai, S. Stobart, N. Parrington, and J. B. Thompson. Iterative design and testing within the software development life cycle. *Software Quality Journal*, (4):295–310.
29. R. P. van der Krogt, M. M. de Weerdt, N. Roos, and C. Witteveen. Multiagent planning through plan repair. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-05)*, pages 1337–1338. ACM press, 2005.
30. H. Warnquist, J. Kvarnström, and P. Doherty. Planning as Heuristic Search for Incremental Fault Diagnosis and Repair. In *Proceedings of the Scheduling and Planning Applications Workshop (SPARK) at the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.
31. P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct. 1990.
32. M. J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.)*. Wiley, 2009.
33. H. H. Zhuo, Q. Yang, and S. Kambhampati. Action-model based multi-agent plan recognition. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *NIPS*, pages 377–385.