Runtime Verification Through Forward Chaining

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
a.garcez@city.ac.uk

In this paper we present a novel rule-based approach for Runtime Verification of FLTL properties over finite but expanding traces. Our system exploits Horn clauses in implication form and relies on a forward chaining-based monitoring algorithm. This approach avoids the branching structure and exponential complexity typical of tableaux-based formulations, creating monitors with a single state and a fixed number of rules. This allows for a fast and scalable tool for Runtime Verification: we present the technical details together with a working implementation.

1 Introduction

We are designing a framework for combining runtime verification and learning in connectionist models to improve the verification of compliance of systems based on business processes. By adapting formal specifications of such systems to include tolerable soft-violations occurring in real-practice to optimise the systems, we want to obtain a more realistic representation of compliance. Adaptation is the recent trend in Process Mining [1]: the goal is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs readily available in todays (information) systems. Within this wider framework, this paper focuses on the introduction of a novel monitoring system, RuleRunner, built as a set of Horn clauses in implication form and exploiting forward chaining to perform runtime verification tasks. A RuleRunner system can be encoded in a recurrent neural network exploiting results from the Neural-Symbolic Integration [8] area, but this is outside the scope of this paper.
This paper is structured as follows: Section 2 introduces background and related work, while Section 3 provides a technical introduction of our rule system. Section 4 provides experimental results and Section 5 ends the paper with final considerations and directions for future work.

2 Background and Related Work

2.1 Horn Clauses and Chaining

A Horn clause [10] is a clause which contains at most one positive literal. The general format of such a clause is thus as follows:

$$\neg\alpha_1 \vee .. \vee \neg\alpha_n \vee \beta$$

This may be rewritten as an implication:

$$(\alpha_1 \wedge .. \wedge \alpha_n) \rightarrow \beta$$

where β is called *head* and $(\alpha_1 \wedge .. \wedge \alpha_n)$ is called *body*. The two formulations are equivalent, and usually the former is called *disjunctive form* and the latter *implication form*. Horn clauses are used for knowledge

Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko,
Valerio Senni (Eds.): First Workshop on
Horn Clauses for Verification and Synthesis (HCVS 2014)
EPTCS 169, 2014, pp. 68–81, doi:10.4204/EPTCS.169.8

representation and automatic reasoning; in particular, inference with Horn clauses can be done through backward or forward chaining. Backward chaining algorithms are goal-driven approaches that work their way from a given goal or query; it is implemented in logic programming (e.g. in Prolog) by SLD resolution [17]. Forward chaining is a data-driven approach that starts with the available data and uses inference rules to extract more data until a goal is reached; it is a popular implementation strategy for production rule systems [12].

2.2 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 [13]. 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 [16], there are many semantics for finite traces: FLTL [14], RVLTL [3], LTL3 [4], LTL$\pm$ [7] 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 [3], 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 $W\phi$) w.r.t. u at position $i < n$, denoted by $[u, i \vDash \psi]$, is an element of $\mathbb{B}$ and is defined as follows:

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

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 dealing with finite traces. Suppose to monitor $\phi = \Box a$ over a trace t, where a is observed in all cells: we have that $[t \vDash \phi]$ 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 [13]. 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, and RuleRunner is inspired by this powerful idea. However, RuleR rules may contain disjunctions in the head and therefore do not correspond to Horn clauses. Furthermore, RuleR creates alternative *observations expectations*, and therefore the application of forward-chaining inference mechanisms on a RuleR system creates a branching, Kripke-like *possible world structure* [11]. We focus on FLTL and encode each formula in a system of rules that correspond to Horn clauses and therefore allow to apply forward-chaining inference algorithms. The next section will describe the difference in the two approaches in more detail.

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:

$$\phi ::= true \mid \ a \mid !a \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi U \phi \mid X\phi \mid W\phi \mid \Diamond\phi \mid \Box\phi \mid END$$

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: function RR-MONITORING(φ,trace t)
2:     Build a monitor RR_φ encoding φ
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
6:         if φ is verified or falsified then
7:             return SUCCESS or FAILURE respectively
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 into future ones.

3.1 Building the rule system

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.*

Throughout this paper we mostly use the terms *State* and *rule*, as they are used in the Runtime Verification area. However, our rules correspond to Horn clauses in implication form, and what we call *State* corresponds to a *knowledge base*.

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 $R[\phi],[\psi^1]V,\ldots,[\psi^n]V,\rightarrow[\phi]V$ 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, $R[\Diamond\psi],[\psi]T\rightarrow[\Diamond\psi]T$ reads as *if $\Diamond\psi$ is being monitored and ψ holds, then $\Diamond\psi$ is true*.

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 (line 9 in Algorithm 1). For instance, $[\Diamond\psi]?\rightarrow R[\Diamond\psi],R[\psi]$ 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*.

Evaluation rules are Horn clauses in implication form. Reactivation rules usually have several positive conjuncts in the head, and therefore a reactivation rule $A\rightarrow\beta_1,..\beta_n$ (where $A=\alpha_1,..,\alpha_m$) can be rewritten as n separate Horn clauses $A\rightarrow\beta_1,..,A\rightarrow\beta_n$. Having different rules with the same head is something to handle with care in case of backward chaining, as many inferential engines implement a depth-first search and therefore the order of these rules impacts on the result. This is not the case when applying forward chaining, as for all rules, if all the premises of the implication are known, then its conclusion is added to the set of known facts.

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:

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

The disjunction in the head of the RuleR rule corresponds to the additional constraints in the body 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 [15]) 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. Table 1 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 $\phi\vee\psi$ (highlighted cell in Table 1), $?_L$ 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 $\vee_L$ is a unary operator. An example for this is monitoring $\Diamond b\vee a$ 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.

$\vee$	T	$?$	F
T	T	T	T
$?$	T	$?$	$?$
F	T	$?$	F

$\vee_B$	T	$?$	F
T	T	T	T
$?$	T	$?_B$	$?_L$
F	T	$?_R$	F

$\vee_L$	
T	T
$?$	$?_L$
F	F

$\vee_R$	
T	T
$?$	$?_R$
F	F

Table 1: truth table (left) and evaluation tables (right) for $\vee$

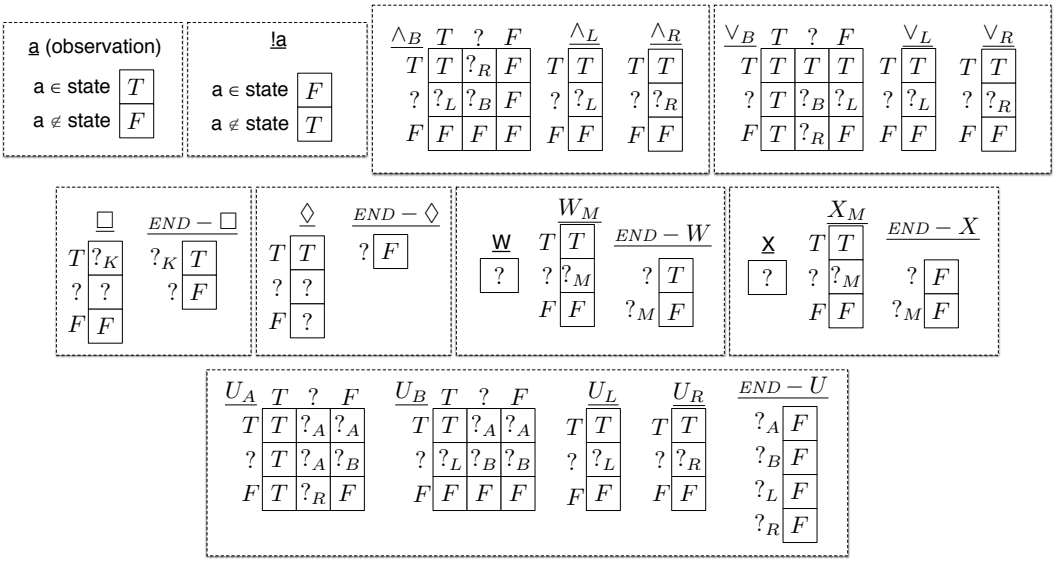

Figure 1: Evaluation tables

The complete set of evaluation tables is reported in Fig. 1, 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

function INITIALISE(ϕ)
  $op \leftarrow$ main operator
  ▷ Apply recursively to subformula(e)
  if $op \in \{\Box, \Diamond, X, W\}$ **then**
    $\langle R_E^1, R_R^1, S^1 \rangle \leftarrow$ Initialise(ψ^1)
    $R_E \leftarrow R_E^1$;
    $R_R \leftarrow R_R^1$;
  else if $op \in \{\vee, \wedge, U\}$ **then**
    $\langle R_E^1, R_R^1, S^1 \rangle \leftarrow$ Initialise(ψ^1)
    $\langle R_E^2, R_R^2, S^2 \rangle \leftarrow$ Initialise(ψ^2)
    $R_E \leftarrow R_E^1 \cup R_E^2$; $R_R \leftarrow R_R^1 \cup R_R^2$;
  else
    $R_E \leftarrow \emptyset$; $R_R \leftarrow \emptyset$;
  end if
  ▷ Compute and add evaluation rules for main operator
  Cells ←op's-evaluation-tables
  for all cell ∈ Cells **do**
    Convert cell to single rule r_e, substituting formula names
    $R_E \leftarrow R_E \cup r_e$
  end for
  if ϕ-is-main-formula **then**
    $R_E \leftarrow R_E \cup ([\phi]T \rightarrow SUCCESS)$
    $R_E \leftarrow R_E \cup ([\phi]F \rightarrow FAILURE)$
    $R_E \leftarrow R_E \cup ([\phi]? \rightarrow REPEAT)$
  end if
  ▷ Compute initial state for this subsystem
  if $op = a$ **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 $op = U$ **then** $S \leftarrow S^1 \cup S^2 \cup R[\phi]A$
  else if $op \in \{\Box, \Diamond\}$ **then** $S \leftarrow S^1 \cup R[\phi]$
  else if $op \in \{X, W\}$ **then** $S \leftarrow R[\phi]$
  end if
  ▷ Compute and add reactivation rules for main operator
  if $op \in \{\vee, \wedge\}$ **then** $R_R \leftarrow R_R \cup ([\phi]?Z \rightarrow R[\phi]?Z)$, for $Z \in L, R, B$
  else if $op = U$ **then** $R_R \leftarrow R_R \cup ([\phi]?Z \rightarrow R[\phi]?Z, S^1, S^2)$, for $Z \in A, B, L, R$
  else if $op \in \{\Box, \Diamond\}$ **then** $R_R \leftarrow R_R \cup ([\phi]? \rightarrow R[\phi], S^1)$
  else if $op \in \{X, W\}$ **then** $R_R \leftarrow R_R \cup ([\phi]? \rightarrow R[\phi]M, S^1) \cup ([\phi]?M \rightarrow R[\phi]M)$
  end if
  ▷ Return computed system
  return $\langle R_E, R_R, S \rangle$
end function

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 $\phi \vee \psi$, 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), and therefore the size of the rule set is linear w.r.t. the structure of the encoded formula ϕ. The obtained rule set does not change at runtime nor when monitoring new traces.

3.2 Verification through Forward Chaining

A RuleRunner rule system RR_ϕ encodes a FLTL formula ϕ in a set of rules. RR_ϕ can be used to check whether a given trace t verifies or falsifies ϕ. Given a set of Horn clauses in rule form R and a set of atoms

A, let the $FC(\cdot)$ (Forward Chaining) function be:

$$FC(R,A) = \{\beta \mid (A_i \rightarrow \beta) \in R \ \wedge \ A_i \subseteq A\}$$

Algorithm 3 describes how RuleRunner exploits forward chaining to perform a runtime verification task.

Algorithm 3 Runtime Verification using RR_ϕ

```
1: function NN-MONITOR(φ,trace t)
2:     Create RR_φ = ⟨R_R, R_E, S⟩ encoding φ (Algorithm 2)
3:     while new observations exist in t do
4:         S' ← S ∩ obs
5:         while S ≠ S' do
6:             S = S'
7:             S' ← S ∩ FC(S, R_E)
8:         end while
9:         if S contains SUCCESS (resp.FAILURE) then
10:            return return SUCCESS (resp.FAILURE)
11:        end if
12:        S ← FC(S, R_R)
13:    end while
14: end function
```

At the beginning, the rule system RR_ϕ is created. The monitoring loop iterates until *SUCCESS* or *FAILURE* is computed, and the FLTL semantics guarantees this happen in the last cell, if reached. At the beginning of each iteration (corresponding to the monitoring of a cell), the initial state S contains a set of rule names corresponding to the subformulae to be checked in that cell. The observations of that cell are then added to the state of the system, and the state is incrementally expanded by means of forward chaining using the evaluation rules (line 7). This corresponds to computing the truth values of all subformulae of ϕ in a bottom-up way, from simple atoms to ϕ itself. If the monitoring did not compute a final verdict (*SUCCESS*/*FAILURE*), the state for the next cell is computed with a single application of $FC(\cdot)$ using the reactivation rules (line 12). Note that in this case the state is not expanded, as only the output of the forward chaining is stored ($S' \leftarrow S \cap FC(S,R_E)$ vs $S \leftarrow FC(S,R_R)$). This is used to *flush* all the previous truth evaluation, which are to be computed from scratch in the new cell.

During the runtime verification, for each cell, the $FC(\cdot)$ function is applied to the initial observations until the transitive closure of all evaluation rules is computed. The number of applications depends linearly on the encoded formula ϕ: at each iteration the truth values of new subformulae are added, proceeding bottom-up from atoms to ϕ. For instance, if $\phi = a \vee \Diamond b$, the first iteration would compute the truth values for a and b, the second would add to the state the truth evaluation for $\Diamond b$, and finally the third one would compute the truth value of ϕ in the current cell. Therefore, for each cell the number of iterations of $FC(\cdot)$ is linear w.r.t. the structure of ϕ. Each application of $FC(\cdot)$ depends on the number of rules and is again linear w.r.t. the structure of ϕ, as stated in the previous subsection. This would suggest a quadratic complexity. However, in our implementation, (for each cell of the trace) the system goes through all rules exactly once. This is obtained 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 [6]).

As an example, consider the formula $\phi = a \vee \Diamond b$ and the trace $t = [c - a - b, d - b, END]$ (dashes separate cells and commas separate observations in the same cell). 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, for first, 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 $[c - a - b, d - b, END]$

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.

3.3 Semantics

RuleRunner implements the FLTL [14] semantics; however, there are two main differences in the approach. Firstly, FLTL is based on rewriting judgements, and it has no constraints over the accessed cells, while RuleRunner is forced to complete the evaluation on a cell before accessing the next one. Secondly, FLTL proceeds top-down, decomposing the property and then verifying the observations; RuleRunner

propagates truth values bottom up, from observations to the property. In order to show the correspondence between the two formalisms, we introduce the map function:

$$map : \text{Property} \rightarrow \text{FLTL judgement}$$

The *map* function translates the state of a RuleRunner system into a FLTL judgement, analysing the state of the RuleRunner system monitoring ϕ. Since $\Box$ and $\Diamond$ are derivate operators and they don't belong to FLTL specifications, we omit them from the discussion in this section.

function MAP(ϕ, State,index)
  if $SUCCESS \in$ State **then return** $\top$
  else if $FAILURE \in$ State **then return** $\bot$
  else if $[\phi]T \in$ State **then return** $\top$
  else if $[\phi]F \in$ State **then return** $\bot$
  else if $[\phi]?S \in$ State **then** $aux \leftarrow S$
  else find $R[\phi]S \in$ State; $aux \leftarrow S$
  end if
  if $\phi = a$ **then**
    return $[u, index \models a]_F$
  else if $\phi = !a$ **then**
    return $[u, index \models \neg a]_F$
  else if $\phi = \psi^1..\psi^2$ *and* $aux = L$ **then**
    return $map(\psi^1)$
  else if $\phi = \psi^1..\psi^2$ *and* $aux = R$ **then**
    return $map(\psi^2)$
  else if $\phi = \psi^1 \vee \psi^2$ *and* $aux = B$ **then**
    return $map(\psi^1) \sqcup map(\psi^2)$
  else if $\phi = \psi^1 \wedge \psi^2$ *and* $aux = B$ **then**
    return $map(\psi^1) \sqcap map(\psi^2)$
  else if $\phi = \psi^1 U \psi^2$ *and* $aux = A$ **then**
    return $map(\psi^2) \sqcup (map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2))))$
  else if $\phi = \psi^1 U \psi^2$ *and* $aux = B$ **then**
    return $map(\psi^2) \sqcap (map(X(\psi^1 U \psi^2)))$ next
  else if $\phi = X\psi$ *and* $aux \neq M$ **then**
    return $[u, index \models X\psi]_F$
  else if $\phi = W\psi$ *and* $aux \neq M$ **then**
    return $[u, index \models \bar{X}\psi]_F$
  else if $(\phi = X\psi$ *or* $\phi = W\psi)$ *and* $aux = M$ **then**
    return $map(\psi)$
  end if
end function

The following table reports a simple example of an evolution of a RuleRunner step and the corresponding value computed by *map*. Let the property be $a \vee Xb$ and the trace be $u = [b - b]$. The index is incremented when the reactivation rules are fired.

Theorem 1 *For any well-formed FLTL formula ϕ over a set of observations, and for every finite trace u, for every intermediate state s_i in RuleRunner's evolution over u there exist a valid rewriting r_j of*

State	$map(a \vee Xb)$
$R[a], R[Xb], R[a \vee Xb]B$	$[u,0 \models a]_F \sqcup [u,0 \models Xb]_F$
$R[a], R[Xb], R[a \vee Xb]B, b$	$[u,0 \models a]_F \sqcup [u,0 \models Xb]_F$
$R[a], R[Xb], R[a \vee Xb]B, b, [a]F$	$\bot \sqcup [u,0 \models Xb]_F$
$R[a], R[Xb], R[a \vee Xb]B, b, [a]F, [b]?M$	$\bot \sqcup [u,0 \models Xb]_F$
$R[a], R[Xb], R[a \vee Xb]B, b, [a]F, [b]?M, [a \vee Xb]?R$	$[u,0 \models Xb]_F$
$R[b], R[Xb]M, R[a \vee Xb]R$	$[u,1 \models b]_F$
$R[b], R[Xb]M, R[a \vee Xb]R, b$	$[u,1 \models b]_F$
$R[b], R[Xb]M, R[a \vee Xb]R, b, [b]T$	$\top$
$R[b], R[Xb]M, R[a \vee Xb]R, b, [b]T, [Xb]T$	$\top$
$R[b], R[Xb]M, R[a \vee Xb]R, b, [b]T, [Xb]T, [a \vee Xb]T$	$\top$
SUCCESS	$\top$

Table 2: The *map* function

$[u,0 \models \phi]_F$ *such that* $map(\phi) = r_j$. *In other words, RuleRunner's state can always be mapped onto an FLTL judgement over* ϕ.

Proof 1 *The proof proceeds by induction on* ϕ*:*

- $\phi = a$
  If the formula is a simple observation, then the initial state is $R[a]$, *and* $map(R[a]) = [u,0 \models a]_F$. *Adding observation to the state does not change the resulting FLTL judgement. If a is observed, RuleRunner will add* $[a]T$ *to the state, and this will be mapped to* $\top$. *If a is not observed, RuleRunner will add* $[a]F$ *to the state, and this will be mapped to* $\bot$. *So for this simple case, the evolution of RuleRunner's state corresponds either to the rewriting* $[u,0 \models a]_F = \top$ *(if a is observed) or to the rewriting* $[u,0 \models a]_F = \bot$ *(if a is not observed).*

- $\phi = !a$
  This case is analogous tho the previous one, with opposite verdicts.

- $\phi = \psi^1 \vee \psi^2$
  By inductive hypothesis, a RuleRunner system monitoring ψ^1 *always corresponds to a rewriting of* $[u,i \models \psi^1]$. *The same holds for* ψ^2. *Let* $\langle R^i_R, R^i_E, S^i \rangle$ *be RuleRunner system monitoring the subformula* ψ^1, *with* $i \in \{1,2\}$. *A RuleRunner system encoding* ϕ *includes* R^1 *and* R^2 *rules and specific rules for* $\psi^1 \vee \psi^2$ *given the truth values of* ψ^1 *and* ψ^2. *The initial state is therefore* $R[\psi^1 \vee \psi^2] \cup S^1 \cup S^2$, *and this is mapped to* $map(S^1) \sqcup map(S^2)$. *By inductive hypothesis, this is a valid FLTL judgement. In each iteration, as long as the truth value of* $\psi^1 \vee \psi^2$ *is not computed, the state is mapped on* $map(S^1) \sqcup map(S^2)$. *When the propagation of truth values reaches* $\psi^1 \vee \psi^2$, *the assigned truth value mirrors the evaluation table for the disjunction. If either* ψ^1 *or* ψ^2 *is true, then* ϕ *is true, and* $map(\phi) = \top$. *This corresponds to the valid rewriting* $map(S^1) \sqcup map(S^2) = \top$, *given that we are considering the case in which there is a true* ψ^i*:* $[\psi^i]T$ *belongs to the state and* $map(\psi^1) = \top$. *The false-false case is analogous. In the* $?_B$ *case, the mapping is preserved, and this is justified by the fact that both* ψ^1 *and* ψ^2 *are undecided in the current cell, therefore* $map(\psi^i) \neq \top, \bot$, *therefore* $map(\psi^1) \sqcup map(\psi^2)$ *could not be simplified. In the* $?_L$ *case, we have that* $[\psi^2]F$, *therefore* $map(\psi^2) = \bot$. *The FLTL rewriting is* $map(\psi^1) \sqcup map(\psi^2) = map(\psi^1)$, *and*

this is a valid rewriting since $map(\psi^1) \sqcup map(\psi^2) = map(\psi^1) \sqcup \bot = map(\psi^1)$. *The* $?_R$ *case is symmetrical.*

- $\phi = \psi^1 \wedge \psi^2$
  Same as above, with the evaluation table for conjunction on the RuleRunner side and the $\sqcap$ *operator on the FLTL judgement side.*
- $\phi = X\psi$
  A RuleRunner system encoding $X\phi$ *has initial state* $R[X\phi]$, *which is mapped on* $[u,0 \models X\psi]_F$. *Then, if the current cell is the last one,* $R[X\phi]$ *evaluates to* $[X\phi]F$, *and the corresponding FLTL judgement is* $\bot$. *If another cell exists,* $R[X\phi]$ *evaluates to* $[X\phi]?$ *(with the same mapping). When the reactivation rules are triggered,* $[X\phi]?$ *is substituted by* $R[X\psi]M, R[\psi]$. *Over this state,* $map(X\psi) = map(\psi)$, *and the index is incremented since reactivation rules were fired. Therefore, the FLTL rewriting is* $[u,i \models X\psi] = [u,i+1 \models \psi]$, *and this is a valid rewriting.*
- $\phi = W\psi$
  This case is like the previous, but if the current cell is the last then $R[W\psi]$ *evolves to* $[W\psi]T$; *the mapping is rewritten from* $[u,i \models W\psi]$ *to* $\top$, *and this is a valid rewriting if there is no next cell.*
- $\phi = \psi^1 U \psi^2$
  The initial RuleRunner system includes rules for ψ^1, ψ^2 *and for the U operator. As long as* $R[\psi^1 U \psi^2]A$ *is not evalued,* $map(\psi^1 U \psi^2) = map(\psi^2) \sqcup (map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2))))$, *that is, the standard one-step unfolding of the 'until' operator as defined in FLTL. When a truth value for the global property is computed, there are several possibilities. The first one is that* ψ^2 *is true and* $\psi^1 U \psi^2$ *is immediately satisfied. RuleRunner adds* $[\psi^1 U \psi^2]T$ *to the state and* $map(\phi) = \top$; *this corresponds to the rewriting* $map(\psi^2) \sqcup (map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2)))) = \top \sqcup (map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2)))) = \top$, *which is a valid rewriting. The case for* $[\psi^1]F$ *and* $[\psi^2]F$ *is analogous. The* $?_A$ *case means that the evaluation for the until is undecided in the current trace, and is mapped on the standard one-step unfolding of the until operator in FLTL. The* $?_B$ *case implicitly encode the information that 'the until cannot be trivially satisfied anymore', and henceforth the FLTL mapping is* $map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2)))$. *The cases for* $?_L$ *and* $?_R$ *have the exact meaning they had in the disjunction and conjunction cases. For instance, if* $[\psi^1]F$ *and* $[\psi^2]?$, *RuleRunner adds* $[\psi^1 U \psi^2]?R$ *to the state, and for the obtained state* $map(\phi) = map(\psi^2)$. *The sequence of FLTL rewriting is* $map(\psi^2) \sqcup (map(\psi^1) \sqcap (map(X(\psi^1 U \psi^2)))) = map(\psi^2) \sqcup (\bot \sqcap (map(X(\psi^1 U \psi^2)))) = map(\psi^2) \sqcup \bot = map(\psi^2)$.

Corollary 1 *RuleRunner yields a FLTL verdict.*

Proof 2 *RuleRunner is always in a state that can be mapped on a valid FLTL judgement; therefore, when a binary truth evaluation for the encoded formula is given, this is mapped on the correct binary evaluation in FLTL. But since for such trivial case the map function corresponds to an identity, the RuleRunner evaluation is a valid FLTL judgement. The fact that RuleRunner yields a binary verdict is guaranteed provided that the analysed trace is finite, thanks to end-of-trace rules.*

4 Experiments

In order to test the scalability of our system, we tested our prototype against several properties and traces: in this paragraph we report a simple set of experiments and results. These tests involve three FLTL formulae, respectively $\phi^1 = \Diamond a$, $\phi^2 = \Box((a \vee b) \vee (c \vee d))$ and $\phi^3 = \Diamond((a \wedge Xb) \vee (c \wedge Wd))$. We encoded

$\phi^{1,2,3}$ in three RuleRunner systems and used them to monitor traces randomly generated using the Latin alphabet as set of observations. Note that each monitoring process can terminate before the end of the trace (e.g. trivially, if a is observed while monitoring $\Diamond a$); we measured the time required to actually monitor a given number of cells.

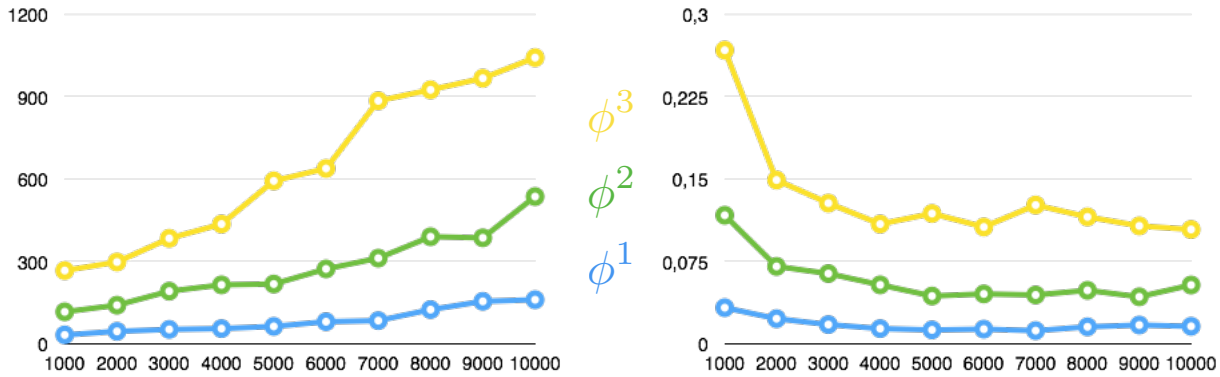

Figure 2: Absolute (left) and averaged (right) performance of monitors encoding $\phi^{1,2,3}$

Figure 2 shows the time required, for the three rule systems, to monitor an increasing number of cells. In both subfigures, the *x-axis* represent the number of monitored cells and the *y-axis* a time measurement in milliseconds (*ms*). The three curves, in both subfigures, correspond to the three monitors for $\phi^1 = \Diamond a$, $\phi^2 = \Box((a \vee b) \vee (c \vee d))$ and $\phi^3 = \Diamond((a \wedge Xb) \vee (c \wedge Wd))$. Figure 2(A) reports total times and Figure 2(B) reports average monitoring time per cell. The trends show how the monitoring time scales w.r.t. the number of the traces; the decreasing of average times in the curves of Figure 2(B) is due to the overhead to compute the rule system becoming less relevant when averaging with a larger number of cells. These experiment can be replicated with the tool available at www.di.unito.it/∼perotti/RuleRunner.jnlp.

5 Conclusions and Future Work

In this paper we present RuleRunner, a rule-based runtime verification system that exploits Horn clauses in implication form and forward chaining to perform a monitoring task. RuleRunner is a module in a wider framework that includes the encoding of the rule system in a neural network, the exploitation of GPUs to improve monitoring performances (as computation in neural networks boils down to matrix-based operations) and the adoption of machine learning algorithms to adapt the encoded property to the observed trace. Our final goal is the development of a system for scalable and parallel monitoring and capable to provide a description of patterns that falsified the prescribed temporal property. The applications of this frameworks spans from multi-agent systems (where a system designer may want to use an agent's unscripted solution to a problem as a benchmark for all other agents [9]) to security (where a security manager may want to correct some false positives when monitoring security properties [5]).
Concerning RuleRunner, a future direction of work is to create rule systems for other finite-trace semantics. For instance, we conjecture that removing all rules with *END* would be a valid starting point for the development of a rule system for LTL3; the rule systems for FLTL and LTL3 will then be used to build a rule system for RVLTL. A second direction of future work will be to modify RuleRunner in such a way to use external forward chaining tools for the monitoring (as we use our own inference engine), such as the Constraint Handling Rules extension included in several Prolog implementations.

6 Bibliography

References

[1] Wil M. P. van der Aalst et.al.: *Process Mining Manifesto*. In: *Procs of Business Process Management Workshops 2011*, pp. 169–194. Available at http://dx.doi.org/10.1007/978-3-642-28108-2_19.

[2] Howard Barringer, David E. Rydeheard & Klaus Havelund (2010): *Rule Systems for Run-time Monitoring: from Eagle to RuleR*. *Journal of Logic and Computation* volume 20, pp. pages 675–706. Available at http://dx.doi.org/10.1093/logcom/exn076.

[3] Andreas Bauer, Martin Leucker & Christian Schallhart: *The Good, the Bad, and the Ugly, But How Ugly Is Ugly?* In: *Procs. of Runtime Verification 2007*, pp. 126–138. Available at http://dx.doi.org/10.1007/978-3-540-77395-5_11.

[4] Andreas Bauer, Martin Leucker & Christian Schallhart: *Monitoring of Real-Time Properties*. In: *Procs. of Foundations of Software Technology and Theoretical Computer Science 2006*, pp. 260–272. Available at http://dx.doi.org/10.1007/11944836_25.

[5] David Breitgand, Maayan Goldstein & E. H. Shehory (2011): *Efficient Control of False Negative and False Positive Errors with Separate Adaptive Thresholds*. *Network and Service Management, IEEE Transactions on* 8, pp. 128–140. Available at http://dx.doi.org/10.1109/TNSM.2011.020111.00055.

[6] Doron Drusinsky: *The Temporal Rover and the ATG Rover*. In: *Procs. of the International Workshop on SPIN Model Checking and Software Verification 2000*, pp. 323–330. Available at http://dx.doi.org/10.1007/10722468_19.

[7] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac & David Van Campenhout: *Reasoning with Temporal Logic on Truncated Paths*. In: *Procs. of Computer-Aided Verification 2003*, pp. 27–39. Available at http://dx.doi.org/10.1007/978-3-540-45069-6_3.

[8] Artur S. d'Avila Garcez & Gerson Zaverucha (1999): *The Connectionist Inductive Learning and Logic Programming System*. *Applied Intelligence* volume 11, pp. pages 59–77. Available at http://dx.doi.org/10.1023/A:1008328630915.

[9] Christopher D. Hollander & Annie S. Wu (2011): *The Current State of Normative Agent-Based Systems*. *J. Artificial Societies and Social Simulation* 14, pp. 47–62. Available at http://jasss.soc.surrey.ac.uk/14/2/6.html.

[10] Alfred Horn (1951): *On Sentences Which are True of Direct Unions of Algebras*. *J. Symb. Log.* 16, pp. 14–21. Available at http://dx.doi.org/10.2307/2268661.

[11] Saul A. Kripke (1963): *Semantical Considerations on Modal Logic*. *Acta Philosophica Fennica* 16, pp. 83–94.

[12] Jean-Louis Lassez & Michael J. Maher: *The Denotational Semantics of Horn Clauses as a Production System*. In: *Procs. of the Association for the Advancement of Artificial Intelligence 1983*, pp. 229–231. Available at http://www.aaai.org/Library/AAAI/1983/aaai83-006.php.

[13] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *Journal of Logic and Algebraic Programming* volume 78, pp. pages 293–303. Available at http://dx.doi.org/10.1016/j.jlap.2008.08.004.

[14] Orna Lichtenstein, Amir Pnueli & Lenore D. Zuck: *The Glory of the Past*. In: *in Procs. of Logic of Programs 1985*, pp. 196–218. Available at http://dx.doi.org/10.1007/3-540-15648-8_16.

[15] J. Lukasiewicz (1920): *O logice trjwartosciowej (On Three-Valued Logic)*.

[16] Amir Pnueli: *The temporal logic of programs*. In: *Procs. of the Annual Symposium on Foundations of Computer Science 1977*, pp. 46–57. Available at http://doi.ieeecomputersociety.org/10.1109/SFCS.1977.32.

[17] M. H. Van Emden & R. A. Kowalski (1976): *The Semantics of Predicate Logic As a Programming Language*. *J. ACM* 23, pp. 733–742. Available at http://dx.doi.org/10.1145/321978.321991.