1 Introduction

Runtime verification (RV) of a given correctness property \( \phi \) (often formulated in linear temporal logic LTL \[1\]) aims at determining the semantics of \( \phi \) while executing the system under scrutiny; a monitor is defined as a device that reads a finite trace and yields a certain verdict \[2\]. 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 \[3\], RVTL \[4\], LTL3 \[5\], LTL\(±\) \[6\] 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. There exist several RV systems, and they can be clustered in three main approaches, based respectively on rewriting, automata and rules.

2 RuleRunner

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

**Algorithm 1** Preprocessing and Monitoring Cycle

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

In a nutshell, RuleRunner’s behaviour (Algorithm \[1\]) is the following: in the preprocessing phase, RuleRunner encodes an LTL formula in a rule system.
The rule system verifies the compliance of a trace w.r.t. the encoded property by entering a monitoring loop, composed 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, another monitoring iteration is entered, and -like in RuleR- undecided formulae trigger the reactivation of the corresponding monitoring rule. FLTL semantics guarantees that, if the trace ends, the verdict in the last cell of the trace is binary.

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

**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.

We will define the rules in more detail in the following subsections; however, due to the lack of space, we will omit some technical details in order to keep the focus on the overall approach and the various components’ interaction.

### 2.1 Evaluation and reactivation rules

RuleRunner accepts formulae \( \phi \) generated by the following grammar:

\[
\phi ::= \text{true} | a | \neg a | \phi \lor \phi | \phi \land \phi | \phi^U \phi | X \phi | W \phi | \Diamond \phi | \Box \phi | \text{END}
\]

\( a \) 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), i.e. negation operators pushed inwards to propositional literals and cancellations applied. \( W \) is the weak negation operator. \( \text{END} \) is a special character that is added to the last cell of a trace to mark the end of the input stream.

An evaluation rule for \( \phi \) is formed from an antecedent (body) and a consequent (head). The antecedent is a conjunction of literals, one of them being the rule name \( R[\phi] \), and the others being truth evaluations \( [\psi]V, \) with \( V \in \{ T, F, ? \} \) and \( \psi \) being a subformula of \( \phi \). The consequent is a single atom yielding a truth evaluation for \( \phi \). Reactivation rules have one single atom as antecedent and a conjunction of atoms as consequent. The left-hand side of a reactivation rule is an undecided truth evaluation, the right-hand side a list of rule names. For
example, consider the rules introduced in the previous section:

\[ R[\Box a], [a]T \rightarrow [\Box a]T \]
\[ R[\Box a], [a]F \rightarrow [\Box a]? \]

are evaluation rules (their only output is a truth evaluation), while

\[ [\Box a]? \rightarrow R[a], R[\Box a] \]

is a reactivation rule, binding truth values in one cell to rule activation in the next cell. The concept of rule activation is like introduced in RuleR. A rule is active if the rule name \( R[\phi] \) is in the state \( S \) of the RuleRunner system. For each active rule, if the condition part evaluates to true for the current cell, then the head of the rule is added to the current state. As introduced in Algorithm [1] the verification loop alternately triggers evaluation and reactivation rules: the evaluation rules are used to compute the truth value of the property in the current cell, and the reactivation rules to define what rules are active in the following state.

Each evaluation rule for \( \phi \) corresponds to a single cell of the evaluation table for the main operator of \( \phi \). Evaluation tables are three-valued truth tables (as introduced by Lukasiewicz and Kleene [7]) with further annotations. The tables in Figure 2 give the example for the \( \Box \) and \( \lor \) operators:

\[
\begin{array}{c|ccc}
\Box & T & F & \Box -END \\
T & T & T & T \\
? & ? & ? & ? \\
F & ? & F & ? \\
\end{array}
\]

\[
\begin{array}{c|cccc}
\lor & T & F & \lor_B & \lor_R \\
T & T & T & T & T \\
? & ? & ? & ? & ? \\
F & F & F & ? & ? \\
\end{array}
\]

Fig. 2: Evaluation tables for \( \Box \), truth and evaluation tables for \( \lor \)

The three cells on the left define the run-time behavior of \( \Box \phi \) given the truth value of \( \phi \) in the current cell: \( \Box \phi \) is true if \( \phi \) is true, undecided otherwise. The single cell marked with \( \Box - END \) represents one extra-rule, triggered only if the end of the trace has been reached: in that case, if the truth value of \( \Box \phi \) is undecided, it is mapped to false. Intuitively, this is done since there is 'no future' left to satisfy \( \phi \); moreover, it mirrors the concept of 'forbidden rules' in RuleR.

The right-hand side of Figure 2 reports the three-valued truth value table for \( \lor \) and the evaluation tables for disjunction. \( ?_L, ?_R \) and \( ?_B \) read, respectively, undecided \textit{left}, \textit{right}, \textit{both}. For example, \( ?_L \) means that the future evaluation of the formula will depend on the left disjunct only, since the right one failed:
the $V_R$ is, in fact, a unary operator. This allows the system to 'ignore' future evaluations of the right disjunct and it avoids the need to 'remember' the fact that the right disjunct failed: since that information is time-relevant (the evaluation failed at a given time, but it may succeed in another trace cell), keeping it in the system state and propagating it through time could cause inconsistencies.

The complete set of evaluation tables is reported in Figure 3:

Fig. 3: evaluation tables
Algorithm 2 Generation of rules

1: function Initialise(\(\phi\))
2: \(\alpha\) ← main operator
3: \[
4: \text{if } \alpha \in \{\Box, \Diamond, X, W\} \text{ then } \]
5: \(\langle R_E^1, R_R^1, S^1 \rangle \leftarrow \text{Initialise}(\psi^1)\)
6: \(R_E \leftarrow R_E^1; R_R \leftarrow R_R^1; \)
7: \[
8: \text{else if } \alpha \in \{V, \land, U\} \text{ then } \]
9: \(\langle R_E^2, R_R^2, S^2 \rangle \leftarrow \text{Initialise}(\psi^2)\)
10: \(R_E \leftarrow R_E^2 \cup R_E^2; R_R \leftarrow R_R^2 \cup R_R^2; \)
11: \[
12: \text{else } \]
13: \(R_E \leftarrow \emptyset; R_R \leftarrow \emptyset; \)
14: \[
15: \text{end if } \]
16: \[
17: \text{Convert cell to single rule } r_e, \text{ substituting formula names } \]
18: \(R_E \leftarrow R_E \cup r_e \)
19: \[
20: \text{for all cells } \in \text{Cells do } \]
21: \[
22: \text{Compute initial state for this subsystem } \]
23: \(R_E \leftarrow R_E \cup [(\phi)T \rightarrow \text{SUCCESS}] \)
24: \(R_E \leftarrow R_E \cup [(\phi)F \rightarrow \text{FAILURE}] \)
25: \[
26: \text{end if } \]
27: \[
28: \text{if } \alpha = \text{a then } S \leftarrow R[\text{a}] \]
29: \[
30: \text{else if } \alpha = \text{!a then } S \leftarrow R[\text{!a}] \]
31: \[
32: \text{else if } \alpha \in \{V, \land\} \text{ then } S \leftarrow S^1 \cup S^2 \cup R[\phi]B \]
33: \[
34: \text{else if } \alpha = U \text{ then } S \leftarrow S^1 \cup S^2 \cup R[\phi]A \]
35: \[
36: \text{else if } \alpha \in \{\Box, \Diamond\} \text{ then } S \leftarrow S^1 \cup R[\phi] \]
37: \[
38: \text{else if } \alpha \in \{X, W\} \text{ then } S \leftarrow R[\phi] \]
39: \[
40: \text{end if } \]
41: \[
42: \text{Apply recursively to subformula(e) } \]
43: \[
44: \text{Compute and add evaluation rules for main operator } \]
45: \[
46: \text{Return computed system } \]
47: \[
48: \text{end function } \]

The generation of evaluation and reactivation rules is summarised in Algorithm 2. The algorithm visits the parsing tree in post-order. The system is built incrementally, starting from the system returned by the recursive call(s). As introduced in the previous subsection, a RuleRunner system is defined as \(\langle R_E, R_R, S \rangle\), the symbols meaning evaluation rules, reactivation rules and state respectively. If \(\phi\) is an observation (or its negation), an initial system is created, including two evaluation rules, no reactivation rules and the single \(R[\phi]\) as initial state. If \(\phi\) 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
Consider the formula \( a \lor \lozenge b \) and the trace \([c - a - b, d - b]\). RuleRunner creates the following rule system:

**EVALUATION RULES**

- \( R[a], \) \( a \) is observed \( \rightarrow [a]T \)
- \( R[a], \) \( a \) is not observed \( \rightarrow [a]F \)
- \( R[b], \) \( b \) is observed \( \rightarrow [b]T \)
- \( R[b], \) \( b \) is not observed \( \rightarrow [b]F \)
- \( R(\lozenge b), [b]T \rightarrow [\lozenge b]T \)
- \( R(\lozenge b), [b]F \rightarrow [\lozenge b]F \)
- \( [\lozenge b]? \), \( \{END\} \rightarrow [\lozenge b]F \)
- \( R(a \lor \lozenge b)B, [a]T, [\lozenge b]T \rightarrow [a \lor \lozenge b]T \)
- \( R(a \lor \lozenge b)B, [a]T, [\lozenge b]? \rightarrow [a \lor \lozenge b]T \)
- \( R(a \lor \lozenge b)B, [a]T, [\lozenge b]F \rightarrow [a \lor \lozenge b]T \)
- \( R(a \lor \lozenge b)B, [a]?, [\lozenge b]T \rightarrow [a \lor \lozenge b]T \)
- \( R(a \lor \lozenge b)B, [a]?, [\lozenge b]F \rightarrow [a \lor \lozenge b]T \)
- \( R(\lozenge b), [a]F, [\lozenge b]T \rightarrow [a \lor \lozenge b]T \)
- \( R(\lozenge b), [a]F, [\lozenge b]? \rightarrow [a \lor \lozenge b]?R \)
- \( R(\lozenge b), [a]F, [\lozenge b]F \rightarrow [a \lor \lozenge b]F \)
- \( R(\lozenge b)L, [a]T \rightarrow [a \lor \lozenge b]T \)
- \( R(\lozenge b)L, [a]? \rightarrow [a \lor \lozenge b]?L \)
- \( R(\lozenge b)L, [a]F \rightarrow [a \lor \lozenge b]F \)
- \( R(\lozenge b)R, [\lozenge b]T \rightarrow [a \lor \lozenge b]T \)
- \( R(\lozenge b)R, [\lozenge b]? \rightarrow [a \lor \lozenge b]?R \)
- \( R(\lozenge b)R, [\lozenge b]F \rightarrow [a \lor \lozenge b]F \)
- \( [a \lor \lozenge b]T \rightarrow SUCCESS \)
- \( [a \lor \lozenge b]F \rightarrow FAILURE \)

**REACTIVATION RULES**

- \( [\lozenge b]? \rightarrow R[b], R(\lozenge b) \)
- \( [a \lor \lozenge b]?B \rightarrow R[a \lor \lozenge b]B \)
- \( [a \lor \lozenge b]?L \rightarrow R[a \lor \lozenge b]L \)
- \( [a \lor \lozenge b]?R \rightarrow R[a \lor \lozenge b]R \)

**INITIAL STATE**

\( R[a], R[b], R(\lozenge b), R[a \lor \lozenge b]B \)

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

| state | R[a], R[b], R[\lozenge b]B, R[a \lor \lozenge b] |
|-------|-------------------------------------------------|
| + obs | R[a], R[b], R[\lozenge b]B, R[a \lor \lozenge b], c |
| react | R[b], R[\lozenge b], R[\lozenge b]R |
| state | R[b], R[\lozenge b], R[\lozenge b]R |
| + obs | R[b], R[\lozenge b], R[\lozenge b]R, a |
| eval | b, F, \, \lozenge b?, [a \lor \lozenge b]?R |
| react | R[b], R[\lozenge b], R[\lozenge b]R |
| state | R[b], R[\lozenge b], R[\lozenge b]R |
| + obs | R[b], R[\lozenge b], R[\lozenge b]R, b, d |
| eval | b, T, \, \lozenge bT, [a \lor \lozenge b]T, SUCCESS |
| STOP | PROPERTY SATISFIED |

of the two subformulae. 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 \( i \), \( \phi \) is monitored in cell \( i + 1 \). \( \phi \) is then monitored independently, and the \( X\phi \) (or \( W\phi \)) rule enters a ‘monitoring state’ (suffix M in the table), simply mirroring \( \phi \) truth value and self-reactivating. Finally, \( \Box \) and \( \Diamond \) constantly reactivate themselves and their subformula, unless they are (respectively) falsified and verified at runtime, or forced to true or false when the trace ends.
The behaviour of the runtime monitor is the following:

- In the first cell, $c$ is observed. $a$ is false, $b$ is false, $\Diamond b$ is undecided. The global formula is undecided, but since the trace continues the monitoring goes on.

- In the second cell, $a$ has to be ignored (because the property required it to be observed in the previous cell); 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.

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

## 2.3 Semantics

RuleRunner implements the FLTL [3] 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:

\[
\text{map} : \text{Property} \rightarrow \text{FLTL judgement}
\]

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

```plaintext
def map(\phi, 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
```

7
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 $[u,0] = \phi$ such that $\text{map}(\phi) = r_j$. 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:

- **φ = a**
  If the formula is a simple observation, then the initial state is R[a], and \( \text{map}(R[a]) = [u, 0 \models a] \). 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 \(T\). If a is not observed, RuleRunner will add \([a]F\) to the state, and this will be mapped to \(\perp\). So for this simple case, the evolution of RuleRunner’s state corresponds either to the rewriting \([u, 0 \models a] F = T\) (if a is observed) or to the rewriting \([u, 0 \models a] F = \perp\) (if a is not observed).

- **φ = !a**
  This case is analogous to the previous one, with opposite verdicts.

- **φ = \(ψ^1 \lor ψ^2\)**
  By inductive hypothesis, a RuleRunner system monitoring \(ψ^1\) always corresponds to a rewriting of \([u, i \models ψ^1]\). The same holds for \(ψ^2\). Let \(⟨R_i, R_E, S^i⟩\) 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 \(ψ^1 \lor ψ^2\) given the truth values of \(ψ^1\) and \(ψ^2\). The initial state is therefore \(R[ψ^1 \lor ψ^2] \cup S^1 \cup S^2\), and this is mapped to \(\text{map}(S^1) \cup \text{map}(S^2)\). By inductive hypothesis, this is a valid FLTL judgement. In each iteration, as long as the truth value of \(ψ^1 \lor ψ^2\) is not computed, the state is mapped on \(\text{map}(S^1) \cup \text{map}(S^2)\). When the propagation of truth values reaches \(ψ^1 \lor ψ^2\), 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 \(\text{map}(φ) = T\). This corresponds to the valid rewriting \(\text{map}(S^1) \cup \text{map}(S^2) = T\), given that we are considering the case in which there is a true \(ψ^i\): \([ψ^i]T\) belongs to the state and \(\text{map}(ψ^i) = T\). The false-true 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 \(\text{map}(ψ^i) \neq T, \perp\), therefore \(\text{map}(ψ^1) \cup \text{map}(ψ^2)\) could not be simplified. In the ?\(_L\) case, we have that \([ψ^2]F\), therefore \(\text{map}(ψ^2) = \perp\). The FLTL rewriting is \(\text{map}(ψ^1) \cup \text{map}(ψ^2) = \text{map}(ψ^1)\), and this is a valid rewriting since \(\text{map}(ψ^1) \cup \text{map}(ψ^2) = \text{map}(ψ^1)\). The ?\(_R\) case is symmetrical.

- **φ = \(ψ^1 \land ψ^2\)**
  Same as above, with the evaluation table for conjunction on the RuleRunner side and the \(\sqcap\) operator on the FLTL judgement side.

- **φ = Xψ**
  A RuleRunner system encoding \(Xφ\) has initial state \(R[Xφ]\), which is mapped on \([u, 0 \models Xψ]\). Then, if the current cell is the last one,
\[ R[X\phi] \] evaluates to \([X\phi]F \), and the corresponding FLTL judgement is \( \perp \). 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.

- \( \phi = \psi^1U\psi^2 \)

The initial RuleRunner system includes rules for \( \psi^1, \psi^2 \) and for the \( U \) operator. As long as \( R[\psi^1U\psi^2]A \) is not evaluated, \( map(\psi^1U\psi^2) = map(\psi^2) \sqcup (map(\psi^1) \cap (map(X(\psi^1U\psi^2)))) \), 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 \( \psi^2 \) is true and \( \psi^1U\psi^2 \) is immediately satisfied. RuleRunner adds \([\psi^1U\psi^2]T \) to the state and \( map(\phi) = \top \); this corresponds to the rewriting \( map(\psi^2) \sqcup (map(\psi^1) \cap (map(X(\psi^1U\psi^2)))) = \top \sqcup (map(\psi^1) \cap (map(X(\psi^1U\psi^2)))) = \top \), 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 \( map(\psi^1) \cap (map(X(\psi^1U\psi^2))) \). 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 \([\psi^1U\psi^2]R \) to the state, and for the obtained state \( map(\phi) = map(\psi^2) \). The sequence of FLTL rewriting is \( map(\psi^2) \sqcup (map(\psi^1) \cap (map(X(\psi^1U\psi^2)))) = map(\psi^2) \sqcup (\bot \sqcap (map(X(\psi^1U\psi^2)))) = map(\psi^2) \sqcup \bot = map(\psi^2) \).

**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.

### 2.4 Complexity

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 $\phi$ or $\psi$, 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). Then, during the runtime verification, for each cell of the trace the system goes through all rules exactly once. This is guaranteed by the fact that the rules are added in a precise order, assuring pre-emption for rules evaluating simpler formulae. This is simply implemented by the post-order visit of the parsing tree, as shown in Algorithm (2). This strict ordering among formulae guarantees that, when the set of rules regarding (i.e.) $\phi U \psi$ is considered, both $\phi$ and $\psi$ have been evaluated already, and their evaluations belong to the current state of the system. Therefore, the complexity of the system is inherently polynomial. This complexity 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 kind of distinction is also mentioned in [3]. In general, the exponential nature of many approaches (e.g., Büchi Automata) arises from listing all possible combinations of observations before matching them with the actual ones. We avoid this by computing a single, distributed state, containing only the certain (and therefore single) truth value of every subformula, computed after the observation phase.

3 Prototyping

A Java implementation is available at www.di.unito.it/~perotti/RV13.jnlp. The prototype requires the user to enter a well-formed LTL formula, which is parsed before unlocking the actual verification settings. Traces can be either manually typed in the GUI, randomly generated (after setting some parameters) or loaded from files. The output is binary for the second and third case, and verbose for the first one.

References

[1] Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (1977) 46–57

[2] Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5) (2009) 293–303

[3] Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In Parikh, R., ed.: Logic of Programs. Volume 193 of Lecture Notes in Computer Science., Springer (1985) 196–218
[4] Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In Sokolsky, O., Tasiran, S., eds.: RV. Volume 4839 of Lecture Notes in Computer Science., Springer (2007) 126–138

[5] Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In Arun-Kumar, S., Garg, N., eds.: FSTTCS. Volume 4337 of Lecture Notes in Computer Science., Springer (2006) 260–272

[6] Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: CAV'03. (2003) 27–39

[7] Łukasiewicz, J.: O logice trójwartosciowej. Ruch Filozoficzny 5, ...: (On Three-Valued Logic. In: Borkowski, L. (ed.) Jan Łukasiewicz Selected Works, pp. 87-88. North Holland (1990) (English translation).)

[8] Drusinsky, D.: The temporal rover and the atg rover. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, London, UK, UK, Springer-Verlag (2000) 323–330