International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An Extension of LTL with Rules and its Application to Runtime Verification

Klaus Havelund · **Doron Peled**

Received: date / Accepted: date

Abstract Linear Temporal Logic (LTL) is extensively used in formal methods, in particular in runtime verification (RV) and in model checking. Its propositional version was shown by Wolper [32] to be limited in expressiveness. Several extensions of propositional LTL, which promote the expressive power to that of Büchi automata, have therefore been proposed; however, none of which was, by and large, adopted for formal methods. We present an extension of propositional LTL with rules that is as expressive as these aforementioned extensions. We show deficiencies in expressiveness of *first-order* LTL and present an extension of it with rules, which parallels the propositional version. We show that in this case the expressiveness of the logic increases.

We are motivated by the strive to enhance the utility of RV, in particular in the case where the monitored executions consists of events that carry data. Monitoring executions that carry data has been addressed by numerous authors, and in previous work we provided an algorithm for past-time first-order LTL that uses BDDs to represent relations over data elements. The proposed extension to the logic presented here is, in fact, inspired by classical RV algorithms, hence the extension of the algorithm is seamless. We describe how the DEJAVU RV tool is modified to encapsulate the extended algorithm and provide experimental results.

The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the second author was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

K. Havelund
Jet Propulsion Laboratory, California Institute of Technology, USA
E-mail: klaus.havelund@jpl.nasa.gov

D. Peled
Department of Computer Science, Bar Ilan University, Israel
E-mail: doron.peled@gmail.com

1 Introduction

Runtime verification (RV) [3, 20] refers to the use of rigorous (formal) techniques for *processing* execution traces emitted by a system being observed. The purpose is typically to evaluate the behavior of the observed system. We focus here on *specification-based* runtime verification, where an execution trace is checked against a property expressed using variants of Linear Temporal Logic (LTL).

LTL is a common specification formalism for reactive and concurrent systems. It is often used in model checking and runtime verification. Another formalism that is used for the same purpose is finite automata, often over infinite words. This includes Büchi, Rabin, Street, Muller and Parity automata [31], all having the same expressive power. In fact, model checking of an LTL property is usually performed by first translating the property into a Büchi automaton. The automata formalisms are more expressive than LTL; a classical example by Wolper [32], shows that it is not possible to express in LTL that every even state in the sequence satisfies some proposition p. This has motivated extending LTL in various ways to achieve the same expressive power as Büchi automata: Wolper's ETL [32, 33] uses right-linear grammars, Sistla's QLTL extends LTL with dynamic (i.e., state-dependent, second-order) *quantification over propositions* [30], and the PSL standard [23] extends LTL with regular expressions. However, these and other extensions have not been extensively used for RV.

We present an alternative extension of propositional LTL with *rules*, named RLTL. These rules use auxiliary propositions, not appearing in the model itself; these propositions obtain their values in each state as a function of the prefix of the execution up to and including that state, expressed as a past time temporal formula. This extension fits easily and naturally existing RV algorithms that use incremental summaries of prefixes, e.g., the classical algorithm [21] for past

time LTL (denoted here PLTL), while maintaining also its linear time complexity (in the length of the trace and the size of the formula). In fact, our extension of the logic is inspired by that RV algorithm. The logic RLTL is shown to be equivalent to QLTL and its restriction to past properties RPLTL is equivalent to Büchi automata and second order monadic logic.

Another dimension for expressiveness lies within the difference between propositional logic, which is based on Boolean propositions, and first-order logic, which allows *quantification over data*. First order LTL is referred here as FLTL. We demonstrate the weakness of even FLTL in expressing Wolper's example, relativized to the first-order case, and also in expressing the transitive closure of temporal relations over events. We introduce two alternative ways of extending the expressive power of FLTL, corresponding, respectively, to the propositional logics QLTL and RLTL. The first adds quantification over relations of data, obtaining a logic referred to as QFLTL. The second extension adds rules for the first-order case, and is referred to as RFLTL. Both of these extended logics can express the above examples. We show that for the first-order case, in contrast with the propositional case, the extension of the logic with quantification is more expressive than the extension with rules.

Runtime verification is commonly restricted to *past time* versions of LTL, i.e., to *safety* properties [1], where a violation can be detected and demonstrated after a finite prefix of the execution. We refer to the logic PLTL for the propositional case and to PFLTL for the first-order case; these logics also enjoy elegant RV algorithms, based on the ability to compute summaries of the observed prefixes [18, 21], as opposed to future temporal logics [25]. The RV algorithm, presented here for RPFLTL (the past part of RFLTL) naturally extends the RV algorithm for PFLTL in [18] in the same way that the algorithm we present for RPLTL (the past part of RLTL) extends the RV algorithm in [21] for PLTL.

We present a corresponding extension of the RV tool DEJAVU [17, 18, 19], that realizes the extension of first-order past time LTL with rules (RPFLTL). The DEJAVU tool allows runtime verification of past time first-order temporal logic over infinite domains (e.g., the integers, strings, etc.). It achieves efficiency by using a unique BDD representation of the data part; BDDs correspond to relations over enumerations of the input data, where each enumeration is represented as a Boolean vector. This is a different use of BDDs from the classical model checking representation of sets of Boolean states; e.g., in [6], BDDs are used to represent sets of program locations, and sets of data elements are represented symbolically as formulas.

Our contribution includes defining and studying propositional and first-order LTL logics extended with rules (extensions prefixed with 'R'), and in particular the logic RPFLTL and its implementation. The structure of the paper

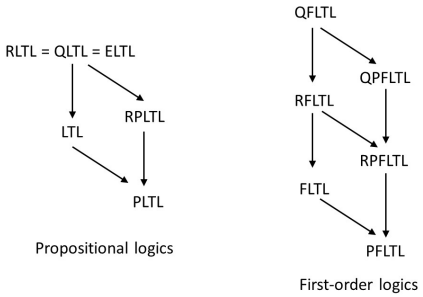


Fig. 1: Logics defined and discussed in this paper.

reflects our step-wise approach by first exploring the problem in the propositional case to form a basic understanding and then by addressing the more interesting first-order case.

Numerous monitoring related expressive logics and systems have been developed over the past decades. In the database community, relations have been added to temporal databases for aggregation [22], calculating functions (sums etc.). Aggregations were also used in the runtime verification tool MONPOLY [4]. Numerous other systems have been produced for monitoring execution traces with data against formal specifications. These include e.g. MOP [27], QEA [29], and LARVA [12], which provide automaton-based data parameterized logics; LOLA [2], which is based on stream processing; BEEPBEEP [15] which is temporal logic-based; and the rule-based LOGFIRE [16]. These systems address the expressiveness issues discussed in this paper in different ways. Our approach differs from earlier such work by taking a starting point in LTL and extending it with rules, implemented using BDDs.

Conventions. As already outlined above, we present several versions of LTL. We name the different versions by prefixing LTL with the following letters. 'P' : restricted to *P*ast-time temporal operators; 'F' : allowing *F*irst-order (static) quantification over data assigned to variables; 'Q' : adding second-order (dynamic) *Q*uantification over propositions/predicates; and finally 'R' : adding *R*ules, our main contribution. The set of logics studied in this paper appear in Figure 1, where each arrow represents inclusion.

2 Propositional LTL

Linear temporal logic [26] has the following syntax:

$$\varphi ::= \mathit{true} \mid p \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid \bigcirc \varphi \mid (\varphi \,\mathcal{U}\, \varphi) \mid \ominus\varphi \mid (\varphi \,\mathcal{S}\, \psi)$$

where p is a proposition from a finite set of propositions P, and $\bigcirc$, $\mathcal{U}$, $\ominus$, $\mathcal{S}$ stand for *next-time*, *until*, *previous-time* and *since*, respectively. A model of an LTL formula is an infinite sequence of states, of the form $\sigma = \sigma[1]\sigma[2], \sigma[3] \ldots$, where

$\sigma[i] \subseteq P$, where $i \geq 1$. These are the propositions that *hold* in that state. LTL's semantics is defined as follows:

- $(\sigma, i) \models \mathit{true}$.
- $(\sigma, i) \models p$ if $p \in \sigma[i]$.
- $(\sigma, i) \models \neg\varphi$ if $(\sigma, i) \not\models \varphi$.
- $(\sigma, i) \models (\varphi \wedge \psi)$ if $(\sigma, i) \models \varphi$ and $(\sigma, i) \models \psi$.
- $(\sigma, i) \models \bigcirc\varphi$ if $(\sigma, i+1) \models \varphi$.
- $(\sigma, i) \models (\varphi \,\mathcal{U}\, \psi)$ if for some j, $j \geq i$, $(\sigma, j) \models \psi$, and for each k, $i \leq k < j$, $(\sigma, k) \models \varphi$.
- $(\sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\sigma, i-1) \models \varphi$.
- $(\sigma, i) \models (\varphi \,\mathcal{S}\, \psi)$ if there exists j, $1 \leq j \leq i$, such that $(\sigma, j) \models \psi$ and for each k, $j < k \leq i$, $(\sigma, k) \models \varphi$.

Then $\sigma \models \varphi$ when $(\sigma, 1) \models \varphi$. We can use the following abbreviations: $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $(\varphi \leftrightarrow \psi) = ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$, $\Diamond\varphi = (\mathit{true}\, \mathcal{U}\, \varphi)$, $\Box\varphi = \neg\Diamond\neg\varphi$, $\mathbf{P}\,\varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ (**P** stands for P*reviously*) and $\mathbf{H}\,\varphi = \neg\mathbf{P}\,\neg\varphi$ (**H** stands for H*istory*).

The expressive power of different versions of propositional LTL is often compared to finite automata over infinite words (Büchi, street, Muler, Parity) and to *monadic* first and second-order logics[1]. Accordingly, we have the following characterizations: LTL is equivalent to monadic first-order logic, and counter-free[2] Büchi automata. For an overview of logic and automata see [31]. Restricting the temporal operators to the *future* operators $\mathcal{U}$ and $\bigcirc$ (and the ones derived from them $\Box$ and $\Diamond$) maintains the same expressive power.

An important subset of LTL, called here PLTL, allows only past temporal operators: $\mathcal{S}$, $\ominus$ and the operators derived from them, **H** and **P**. The past time logic is sometimes interpreted over finite sequences, where $\sigma \models \varphi$ when $(\sigma, |\sigma|) \models \varphi$. It is also a common practice, in particular in RV, to use a PLTL formula, prefixed with a single $\Box$ (always) operator; in this case, *each of the prefixes* has to satisfy φ. This later form expresses *safety* LTL properties [1]. For safety (past) properties, we henceforce assume this interpretation and will not explicitly prefix the formulas with a $\Box$. When PLTL is interpreted over finite sequences, its expressive power is the same as counter-free finite automata and first order monadic logic over finite words. For a safety property, a failure to satisfy it in an execution sequence (a model) can be detected after observing a finite prefix.

Wolper [32] demonstrated that the expressive power of LTL is lacking using the property that

> all the states with even indexes in a sequence satisfy some proposition p.

Note that this is different than stating that p alternates between *true* and *false* on consecutive states.

Extending LTL with dynamic quantification. Adding quantification over propositions, suggested by Sistla in [30], allows writing a formula of the form $\exists q\,\varphi$, where $\exists q$ represents *existential* quantification over a proposition q that can appear in φ. To define the semantics, let $X \subseteq P$ and denote $\sigma \setminus X = (\sigma[1] \setminus X)(\sigma[2] \setminus X)\ldots$, where $\sigma \setminus X$ denotes projecting *out* the propositions in X. The semantics is defined as follows:

- $(\sigma, i) \models \exists q\,\varphi$ if there exists σ' such that $\sigma' \setminus \{q\} = \sigma$ and $(\sigma', i) \models \varphi$.

Universal quantification is defined as $\forall q\,\varphi = \neg\exists q\,\neg\varphi$. This kind of quantification is considered to be *dynamic*, since the quantified propositions can have different truth values depending on the states. It is also called *second-order* quantification, since the quantification establishes the *set* of states in which a proposition has the value *true*. Extending LTL with such quantification, the logic QLTL has the same expressive power as full Büchi automata and monadic second-order logic. In fact, it is sufficient to restrict the quantification to existential quantifiers that appear at the beginning of the formula[3] to obtain the full expressiveness of QLTL [31]. Restricting QLTL to the past modalities, one obtains the logic QPLTL. QPLTL has the same expressive power as regular expressions and finite automata. Wolper's property can be rewritten in QPLTL as:

$$\exists q\,\mathbf{H}((q \leftrightarrow \ominus\neg q) \wedge (q \rightarrow p)) \tag{1}$$

Since $\ominus\varphi$ is interpreted as *false* in the first state of any sequence, regardless of φ, then the truth value of q is *false* in the first state. Then q alternates between even and odd states.

Extending LTL with rules. We introduce another extension of LTL, which we call RLTL. As will be showed later, this extension is natural for runtime verification. We partition the propositions P into *auxiliary propositions* $A = \{a_1, \ldots, a_n\}$ and *basic propositions* B. Only basic propositions can appear in the model.

Definition 1 An RLTL property η has the following form:

$$\psi \textbf{ where } a_j := \varphi_j : j \in \{1, \ldots, n\} \tag{2}$$

where each a_j is a distinct auxiliary proposition from A, ψ is an LTL property and each φ_i is a PLTL property where propositions from A can only occur within the scope of a previous-time ($\ominus$) operator.

[1] The logics are called monadic since they allow relations with *one* parameter that explicitly represents the time; hence instead of occurrences of a proposition p in the temporal logic formulas, the monadic logics have a relation p where $p(i)$ stands for *p holds at time i*. First order monadic logic allows quantifying over time variables, while second order logic allows quantifying over relations.

[2] For a counter-free language, there is an integer n such that for all words x, y, z and integers $m \geq n$ we have that $xy^m z \in L$ if and only if $xy^n z \in L$.

[3] In fact, with some encoding, it is possible to further restrict this to formulas that include only *one* existential quantification at the beginning.

We refer to ψ as the *statement* of η and to $a_j := \varphi_j$ as a *rule* (in text, rules will be separated by commas).

Definition 2 The semantics can be defined as an extension to the above LTL semantics, where for each rule $a_j := \varphi_j$ we define,

- $(\sigma, i) \models a_j$ if $(\sigma, i) \models \varphi_j$.

The constraint that auxiliary propositions appearing in the formulas φ_i must occur within the scope of a $\ominus$ operator is required to prevent conflicting rules, as in $a_1 := \neg a_2$ and $a_2 := a_1$. Wolper's example can be written in RLTL as follows:

$$\Box(q \rightarrow p) \textbf{ where } q := \ominus \neg q \tag{3}$$

where $A = \{q\}$ and $B = \{p\}$. The auxiliary proposition q is used to augment the input sequence such that each *odd* state will satisfy $\neg q$ and each *even* state will satisfy q.

Lemma 1 (Well foundedness of auxiliary propositions) *The values of the auxiliary propositions of an RLTL formula η are uniquely defined in a state by the prefix of the execution up to and including that state.*

Proof. The proof is by induction on the length of prefixes. The value of each auxiliary proposition a_j at the ith state of σ' is defined via a rule $a_j := \varphi_j$, where φ_j is a PLTL formula. Hence it depends on the values of the propositions B in the ith state of σ, and on the values of $A \cup B$ in the previous states of σ'. □

Theorem 1 *The expressive power of RLTL is the same as QLTL.*

Proof. Each RLTL formula η, as defined in (2), is expressible using the following equivalent QLTL formula:

$$\exists a_1 \ldots \exists a_n (\psi \wedge \Box \bigwedge_{1 \leq j \leq n} (a_j \leftrightarrow \varphi_j))$$

For the other direction, one can first translate the QLTL property into a second-order monadic logic formula, then to a deterministic Muller automata [31] and then construct an RLTL formula that holds for the accepting executions of this automaton. The rules of this formula encode the automata states, and the statement describes the acceptance condition of the Muller automaton. □

We define RPLTL as the past version of RLTL, i.e., by disallowing the future time temporal operators in RLTL. As before, every top level formula is interpreted as implicitly being prefixed with a $\Box$ operator, i.e., needs to hold in every prefix. This results in a formalism that is equivalent to Büchi automata, where all the states except one are accepting and where the non-accepting state is a sink. We can use a related, but simpler construction than in Theorem 1 to prove the following:

Lemma 2 *The expressive power of RPLTL is the same as QPLTL.*

Lemma 3 *RPLTL can, with no loss of expressive power, be restricted to the form:*

$$\Box p \textbf{ where } a_j = \varphi_j : j \in \{1, \ldots, n\}$$

with p being one of the auxiliary propositions a_j and each φ_j contains only a single occurrence of the $\ominus$ temporal operator (and the Boolean operators).

In this form, the values of the Boolean variables a_j encode the states of an automaton, and the rules encode the transitions.

3 RV for Propositional Past Time LTL and its Extension

Runtime verification of temporal logic specifications can be designed to concentrate on past time properties. Past time specifications have the important property that one can distinguish when they are violated after observing a finite prefix of an execution. For an extended discussion of the issue of *monitorability*, see e.g., [5, 13]. The RV algorithm for PLTL, presented in [21], is based on the observation that the semantics of the past time formulas $\ominus\varphi$ and $(\varphi \mathcal{S} \psi)$ in the current state i is defined in terms of the semantics of its subformula(s) in the previous state $i-1$. To demonstrate this, we rewrite the semantic definition of the $\mathcal{S}$ operator in a form that is more applicable for runtime verification.

- $(\sigma, i) \models (\varphi \mathcal{S} \psi)$ if $(\sigma, i) \models \psi$ or: $i > 1$ and $(\sigma, i) \models \varphi$ and $(\sigma, i-1) \models (\varphi \mathcal{S} \psi)$.

The semantic definition is recursive in both the length of the prefix and the structure of the property. Thus, subformulas are evaluated based on smaller subformulas, and the evaluation of subformulas in the previous state. The algorithm shown below uses two vectors of values indexed by subformulas: pre, which summarizes the truth values of the subformulas for the execution prefix that ends just *before* the current state, and now, for the execution prefix that ends with the current state.

The *summary* of a property η after some finite sequence of states σ is the collection of values $\mathsf{pre}(\varphi)$ and $\mathsf{now}(\varphi)$ for the subformulas φ of η calculated by the RV algorithm. It is called a summary, since it summarizes the information that the RV algorithm requires to remember after observing a prefix of an execution path. The order of calculating now for subformulas is bottom up, according to the syntax tree.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) :=$ *false*.
2. Observe a new event (as a set of propositions) s as input.
3. Let $\mathsf{pre}(\varphi) := \mathsf{now}(\varphi)$ for each subformula φ.

4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \mathit{true}$.
   - $\mathsf{now}(p) := (p \in s)$
   - $\mathsf{now}(\varphi \wedge \psi) := (\mathsf{now}(\varphi) \wedge \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) := \neg\mathsf{now}(\varphi)$.
   - $\mathsf{now}(\varphi \mathcal{S} \psi) := (\mathsf{now}(\psi) \vee (\mathsf{now}(\varphi) \wedge \mathsf{pre}(\varphi \mathcal{S} \psi)))$.
   - $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.
5. If $\mathsf{now}(\eta) = \mathit{false}$ then report a violation, otherwise goto step 2.

Runtime verification for RPLTL. For RPLTL, we need to add to the above algorithm calculations of $\mathsf{now}(a_j)$ and $\mathsf{now}(\varphi_j)$ for each rule of the form $a_j := \varphi_j$ The corresponding pre entries will be updated as in step 3 above. Because the auxiliary propositions can appear recursively in RPLTL rules, the order of calculation is subtle. To see this, consider for example Formula (3). It contains the definition $q := \ominus\neg q$. We cannot calculate this bottom up in the way we did for PLTL, since $\mathsf{now}(q)$ is not computed yet, and we need to calculate $\mathsf{now}(\ominus\neg q)$ in order to compute $\mathsf{now}(q)$. However, notice that the calculation is not dependent on the value of q to calculate $\ominus\neg q$; in Step 4 above, we have that $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$ so $\mathsf{now}(\ominus\neg q) := \mathsf{pre}(\neg q)$.

We use *mixed evaluation order*. Under this evaluation order, one calculates now as part of Step 4 of the above algorithm in the following order.

a. Calculate $\mathsf{now}(\delta)$ for each subformula δ that appears in φ_j of a rule $a_j := \varphi_j$, but *not* within the scope of a $\ominus$ operator (observe that $\mathsf{now}(\ominus\gamma)$ is set to $\mathsf{pre}(\gamma)$).
b. Set $\mathsf{now}(a_j)$ to $\mathsf{now}(\varphi_j)$ for each j.
c. Calculate $\mathsf{now}(\delta)$ for each subformula δ that appears in φ_j within a rule $a_j := \varphi_j$ *within* the scope of a $\ominus$ operator.
d. Calculate $\mathsf{now}(\delta)$ for each subformula δ that appears in the statement ψ using the calculated $\mathsf{now}(a_j)$.

4 First-Order LTL

We study now a family of first order temporal logics, which allow specifying the properties of sequences with data. The presentation follows the same high level structure used to study the propositional logics in the previous sections, starting with the basic logic and its extensions, followed by RV algorithms.

Assume a finite set of infinite domains[4] $D_1, D_2, \ldots$, e.g., integers or strings. Let V be a finite set of *variables*, with typical instances x, y, z. An *assignment* over the set of variables V maps each variable $x \in V$ to a value from its associated domain $\mathit{domain}(x)$. For example $[x \to 5, y \to \text{“abc”}]$ assigns the values 5 to x and the value “abc” to y.

[4] Finite domains are handled with some minor changes, see [18].

Syntax. The grammar of FLTL is defined as follows, where p denotes a relation, a denotes a constant and x denotes a variable:

$$\varphi ::= \mathit{true} \mid p(a) \mid p(x) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid \bigcirc\varphi \mid (\varphi\ \mathcal{U}\ \varphi) \mid \ominus\varphi \mid (\varphi\ \mathcal{S}\ \varphi) \mid \exists x\, \varphi$$

Additional operators are defined as in the propositional logic. We define $\forall x\, \varphi = \neg\exists x \neg\varphi$. Restricting the modal operators to the past operators ($\mathcal{S}$, $\ominus$ and the ones derived from them) forms the logic PFLTL.

Semantics. A first order *model* σ is a sequence $\sigma = \sigma[1]\sigma[2]\ldots$, where for each $i > 1$, $\sigma[i]$ is a set of relations. The relations in all the states have the same types, i.e., arities and domains. The relation p in the i^{th} state $\sigma[i]$ of σ is $\sigma[i](p)$, hence $\sigma[i](p)(a)$ means that $p(a)$ holds in $\sigma[i]$.

Let $\mathit{free}(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. Let ε be the empty assignment (with no variables). $(\gamma, \sigma, i) \models \varphi$ is defined where γ is an assignment over a set of variables that *includes* $\mathit{free}(\varphi)$, and $i \geq 1$. Let $\gamma[x \mapsto a]$ be the overriding of γ with the binding $[x \mapsto a]$.

- $(\gamma, \sigma, i) \models \mathit{true}$.
- $(\gamma, \sigma, i) \models p(a)$ if $\sigma[i](p)(a)$.
- $(\gamma[x \mapsto a], \sigma, i) \models p(x)$ if $\sigma[i](p)(a)$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma, \sigma, i) \models \varphi$ and $(\gamma, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models \bigcirc\varphi$ if $(\gamma, \sigma, i+1) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\ \mathcal{U}\ \psi)$ if for some j, $j \geq i$, $(\gamma, \sigma, j) \models \psi$ and for each k, $i \leq k < j$, $(\gamma, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\ \mathcal{S}\ \psi)$ if for some j, $1 \leq j \leq i$, $(\gamma, \sigma, j) \models \psi$ and for each k, $j < k \leq i$, $(\gamma, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x\, \varphi$ if there exists $a \in \mathit{domain}(x)$ such that $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

For an FLTL (PFLTL) formula with no free variables, we let $\sigma \models \varphi$ denote $(\varepsilon, \sigma, 1) \models \varphi$. We will henceforce, less formally, use the same symbols both for the relations (semantics) and their representation in the logic (syntax). Note that the letters p, q, r, which were used for representing propositions in the propositional versions of the logic in previous sections, will represent relations in the first-order versions. The quantification over values of variables, denoted by $\exists$ and $\forall$, is *static*, in the sense that they are independent of the state in the execution. We demonstrate that the lack of expressiveness carries over from LTL (PLTL) to FLTL (PFLTL).

Example 1 [A generalization of Wolper's example] Let p and q be unary relations. The specification that we want to monitor is that for each value a, $p(a)$ appears in all the states where $q(a)$ has appeared an even number of times so far (for the odd occurrences, $p(a)$ can also appear, but does not have to appear). To show that this is not expressible in FLTL (and PFLTL), consider models where only one data element a appears. Assume for the contradiction that there is an FLTL

formula ψ that expresses this property. We recursively replace in ψ each subformula of the form $\exists\varphi$ by a disjunction over copies of φ, in which the quantified occurrences of $p(x)$ and $q(x)$ are replaced by p_a and q_a, respectively, or to *false*; *false* represents the Boolean value of $p(x)$ and $q(x)$ for any $x \neq a$, since only $p(a)$ and $q(a)$ appear in the model. For example, $\exists x(q(x)\mathcal{S}p(x))$ becomes $((q_a\,\mathcal{S}\,p_a) \vee (false\,\mathcal{S}\,false))$, which can be simplified to $(q_a\,\mathcal{S}\,p_a)$. Similarly, subformulas of the form $\forall\varphi$ are replaced by conjunctions. This results in an LTL formula that holds in a model, where each $p(a)$ is replaced by p_a and each $q(a)$ is replaced by q_a, iff ψ holds for the original model. Assume further that $q(a)$ holds everywhere (in fact, the relation q was added only to make the example more interesting). Then Wolper's example contradicts the assumption that such a formula exists

Using parametric automata as a specification formalism, as in [14, 20, 27, 29], *can* express this property, where for each value a there is a separate automaton that counts the number of times that $q(a)$ has occurred.

Example 2 [Transitive closure] Consider the property that when $report(y,x,d)$ appears in a state, denoting that process y sends some data d to a process x, there was a chain of process spawns: $spawn(x,x_1)$, $spawn(x_1,x_2)$ $\ldots$ $spawn(x_l,y)$. i.,e., y is a descendent process of x. The required property involves the transitive closure of the relation *spawn*. This property cannot be expressed in FLTL: FLTL can be translated, in a way similar to the standard translation of *LTL*, into monadic first-order logic [31], with explicit occurrences of time variables over the Naturals and the linear order relation $<$ (or $\leq$) between them. The relations will be written with an additional *time* parameter, and the temporal operators are replaced with first-order quantification as in the propositional case. For example, $\Box\forall x\,(p(x) \rightarrow \Diamond q(x))$ will be translated into $\forall x \forall t\,(p(x,t) \rightarrow \exists t'\,(t \leq t' \wedge q(x,t')))$. However, the transitive closure of *spawn* cannot be expressed in the first-order setting. This can be shown based on the compactness theory of first-order logic [11].

Extending FLTL with dynamic quantification. Relations play in FLTL a similar role to propositions in LTL. Hence, in correspondence with the relation between LTL and QLTL, we extend FLTL (PFLTL) with dynamic quantification over auxiliary relations, which do not appear in the model, obtaining QFLTL (and the past-restricted version QPFLTL). The syntax includes $\exists p\varphi$, where p denotes a relation. We also use $\forall p\ \varphi = \neg\exists p\neg\varphi$. The semantics is as follows.

- $(\gamma,\sigma,i) \models \exists q\,\varphi$ if there exists σ' such that $\sigma' \setminus \{q\} = \sigma$ and $(\gamma,\sigma',i) \models \varphi$.

Note that quantification here is dynamic (as in QLTL and QPLTL) since the relations can have different sets of tuples in different states. As an example, consider a formalization of the property in Example 1:

$$\exists r \forall x((r(x) \rightarrow p(x)) \wedge (r(x) \leftrightarrow (q(x) \leftrightarrow \ominus\neg r(x)))) \tag{4}$$

The formula introduces an auxiliary unary relation r over the same type of argument as p and q. For each value a, $r(a)$ flips its truth value from when $q(a)$ holds. Note that $(r(x) \leftrightarrow \neg q(x))$ in $\sigma[1]$ since $\ominus\neg r(x)$ is *false* in that state.

Extending FLTL with rules. We now extend FLTL into RFLTL in a way that was motivated by the propositional extension of LTL (PLTL, respectively) to RLTL (RPLTL, respectively).

Definition 3 An RFLTL property has the following form:

$$\psi \textbf{ where } r_j(x_j) := \varphi_j(x_j) : j \in \{1,\ldots,n\} \tag{5}$$

such that,

1. ψ, the *statement*, is an FLTL formula with no free variables,
2. φ_j are PFLTL formulas with a single[5] free variable x_j,
3. r_j is an auxiliary relation over the same type as x_j. An auxiliary relation r_j can appear within ψ. It can also appear in $\varphi_k(x_k)$ of a *rule* $r_k(x_k) := \varphi_k(x_k)$, but only within the scope of a previous-time operator $\ominus$.

We extend the semantics of FLTL and to RFLTL by adding the following items, per each rule of the form $r_j(x_j) := \varphi_j(x_j)$:

- $(\gamma[x \mapsto a],\sigma,i) \models r_j(x)$ if $(\gamma[x \mapsto a],\sigma,i) \models \varphi(x)$.

Lemma 4 *Each RFLTL formula of the form (5) can be expressed using the following QFLTL property:*

$$\exists r_1 \ldots \exists r_n\,(\psi \wedge \Box \bigwedge_{j\in\{1,\ldots,n\}} (r_j(x_j) \leftrightarrow \varphi_j(x_j))) \tag{6}$$

The logic RPFLTL is obtained by restricting the temporal modalities of RFLTL to the past ones: $\mathcal{S}$ and $\ominus$, and those derived from them.

Lemma 5 (Well foundedness of auxiliary relations) *The auxiliary temporal relations of an RFLTL formula at state i are uniquely defined by the prefix of the execution up to and including that state.*

Proof. By a simple induction on the length of prefixes, similar to Lemma 1. □

The following formula expresses the property described in Example 1, which was shown not to be expressible using FLTL.

$$\forall x\,(r(x) \rightarrow p(x)) \textbf{ where } r(x) = (q(x) \leftrightarrow \ominus\neg r(x)) \tag{7}$$

The property that corresponds to Example 2 can be expressed as:

[5] Again, the definition can be extended to any number of parameters.

$\forall x \forall y \forall d\,(report(y,x,d) \rightarrow spawned(x,y))$
where $spawned(x,y) = ((\ominus spawned(x,y) \vee spawn(x,y)) \vee \exists z\,(\ominus spawned(x,z) \wedge spawn(z,y)))$

It also appears as the property spawning in Figure 5 in the implementation section 6.

Theorem 2 *The expressive power of RPFLTL is strictly weaker than that of QPFLTL.*

The proof of Theorem 2 will be given in the next section, since it will use a recursive argument that is easier to explain given the RV algorithm.

5 RV for Past Time First-Order LTL and its Extension

Runtime verification of FLTL is performed on an input that consists of *events* in the form of tuples of relations. In our notation, the input consists of a sequence of events $\sigma[1]\sigma[2]\ldots$, which we earlier identified with states, where each $\sigma[i]$ consists of relations, with the same type and arity for each i. A typical use of runtime verification restricts the events for each state to a single event; correspondingly, $\sigma[i]$ will be restricted to a relation with a single tuple.

Set Semantics. We provide an alternative *set semantics* for the logic RPFLTL, which is equivalent to the above definition. The set semantics is more directly related to the calculation of the summary values in the RV algorithm that will be presented below, which consist of sets of assignments, represented as relations. Under set semantics, introduced in [18] for PFLTL, and extended here for RPFLTL, $I[\varphi,\sigma,i]$ denotes a set of assignments such that $\gamma \in I[\varphi,\sigma,i]$ iff $(\gamma,\sigma,i) \models \varphi$, where γ is an assignment over a set of variables that contain the free variables in φ. We fix a set of variables V that includes all the variables that appear in the formula φ, and denote by $\mathcal{A}_V$ the set of all possible assignments of domain values to the variables V. Assuming an order between the variables in V, we treat sets of assignments over V as relations, hence allow applying the operators $\bigcup$ (set union) and $\bigcap$ (set intersection) on sets of assignments. The operator $hide(\Gamma,W)$ replaces each assignment (tuple) γ in the relation Γ with the set of assignments that have any possible domain value for the variables in W. Thus, *hide* has the effect of projecting out from Γ the values of the variables W and then completing the assignments to these variables arbitrarily. In the set semantics, we define $I[\varphi,\mathcal{R},i] \subseteq \mathcal{A}_V$ by structural recursion. To simplify the definitions, we add a dummy position $\sigma[0]$ for sequence σ (which starts with $\sigma[1]$), where every formula is interpreted as an empty set. Observe that the values $\emptyset$ and $\mathcal{A}_V$, behave as the Boolean constants 0 and 1, respectively. The set semantics is defined as follows, where $i \geq 1$.

- $I[\varphi,\sigma,0] = \emptyset$.
- $I[true,\sigma,i] = \mathcal{A}_V$.
- $I[p(a),\sigma,i] =$ if $\sigma[i](p)(a)$ then $\{\varepsilon\}$ else $\emptyset$.
- $I[p(x),\sigma,i] = \{\gamma[x \mapsto a] \mid \gamma \in \mathcal{A}_V \wedge \sigma[i](p)(a)\}$.
- $I[(\varphi \wedge \psi),\sigma,i] = I[\varphi,\sigma,i] \bigcap I[\psi,\sigma,i]$.
- $I[\neg\varphi,\sigma,i] = \mathcal{A}_V \setminus I[\varphi,\sigma,i]$.
- $I[(\varphi\ \mathcal{S}\ \psi),\sigma,i] = I[\psi,\sigma,i] \bigcup (I[\varphi,\sigma,i] \bigcap I[(\varphi \mathcal{S} \psi),\sigma,i-1])$.
- $I[\ominus\varphi,\sigma,i] = I[\varphi,\sigma,i-1]$.
- $I[\exists x\,\varphi,\sigma,i] = hide(I[\varphi,\sigma,i],\{x\})$.
- $I[r_j(x),\sigma,i] := I[\varphi_j(x),\sigma,i]$.

The semantics of $(\varphi\ \mathcal{S}\ \psi)$ reflects the following valid equivalence: $(\varphi\ \mathcal{S}\ \psi) \equiv \psi \vee (\varphi \wedge \ominus(\varphi\ \mathcal{S}\ \psi))$. The last item in the semantic definition is related to rules of the form $r_j(x_j) = \varphi_j(x_j)$.

Runtime verification algorithm for PFLTL. We start by describing an algorithm for monitoring PFLTL properties, presented in [18] and implemented in the tool DEJAVU. The basic idea is to represent a set of assignments of data to variables as relations. We enumerate data values appearing in monitored events, as soon as we first see them. We represent enumerations as bit-vectors (i.e., Binary) encodings and represent the relations over the (bit-vector) enumerations rather than data values themselves, where bit vectors for different values are concatenated together. The relations are then represented as BDDs [7]. BDDs were featured in model checking because of their ability to frequently achieve a highly compact representation of Boolean functions. Based on set semantics, our algorithm for the first-order logic is conceptually similar to the propositional case, but where the Boolean values in the summaries are replaced by relations represented as BDDs. The extensive work done on BDDs allowed us to use an optimized public BDD package.

Since we want to be able to deal with infinite domains (where only a finite number of elements may appear in a given observed prefix) and maintain the ability to perform complementation, unused enumerations represent the values that have not been seen yet. In fact, it is sufficient to use one enumeration representing these values per each variable of the LTL formula. We guarantee that at least one such enumeration exists by reserving for that purpose the enumeration $11\ldots11$. We present here only the basic algorithm. For versions that allow extending the number of bits used for enumerations and garbage collection of enumerations, see [17].

When a ground predicate $p(a)$ is observed in the monitored execution, matched with $p(x)$ in the monitored property, a call to the procedure **lookup**(x,a) returns the enumeration of a, based on a lookup in the hash table. If this is the first occurrence of a, then a will be assigned a new enumeration, which will be stored under a. We can use a counter, for each variable x, counting the number of different values appearing so far for x. When a new value appears,

this counter is incremented and converted to a binary (bit-vector) representation. Note, however, that other enumeration generation schemes are possible; our implementation uses a BDD based satisfiability procedure to generate an unused enumeration (this scheme is needed due to the garbage collection performed on BDD enumerations [17]). The function $\mathbf{build}(x, C)$ returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) a for $a \in C$. This BDD is independent of the values assigned to any variable other than x, i.e., they can have any value.

For example, assume that the runtime-verifier sees the input events *open*("a"), *open*("b"), *open*("c"), and assume that it encodes the argument values with 3 bits[6] x_1, x_2, and x_3 to represent the enumerations, with x_1 being the least significant bit. Assume further, using an incremental enumeration, that the value "a" gets mapped to the enumeration $x_3x_2x_1 = 000$ (Natural number 0) and that the value "b" gets mapped to the enumeration $x_3x_2x_1 = 001$ (Natural number 1). That is, $\mathbf{lookup}(x, a) = 000$, and $\mathbf{lookup}(x, b) = 001$. A Boolean representation of the *set* of values $C = \{$"a","b"$\}$ would be equivalent to a Boolean function $(\neg x_2 \wedge \neg x_3)$ that returns 1 for 000 and 001. That is, $\mathbf{build}(x, C)$ is a BDD representation of this Boolean function.

Intersection and union of sets of assignments are translated simply into conjunction and disjunction of their BDD representation, respectively; complementation becomes BDD negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential (universal, respectively) operators, we use the BDD existential (universal, respectively) operators over the Boolean variables that represent (the enumerations of) the values of x. Thus, if B_φ is the BDD representing the assignments satisfying φ in the current state of the monitor, then $\mathbf{exists}(\langle x_1, \ldots, x_k \rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x \varphi$ in the current state. Finally, BDD($\bot$) and BDD($\top$) are the BDDs that return always 0 or 1, respectively.

For first order RV, we will use a summary, consisting of $\mathsf{pre}(\varphi)$ and $\mathsf{now}(\varphi)$ for each subformula φ of the checked property η. For first order temporal logic, these elements are BDDs representing relations. The algorithm for monitoring a formula η is as follows.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) :=$ BDD($\bot$).
2. Observe a new state (as a set of ground predicates) $\sigma[i]$ as input.
3. Let $\mathsf{pre}(\varphi) := \mathsf{now}(\varphi)$ for each subformula φ.
4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) :=$ BDD($\top$).
   - $\mathsf{now}(p(a)) :=$ if $\sigma[i](p)(a)$ then BDD($\top$) else BDD($\bot$).
   - $\mathsf{now}(p(x)) := \mathbf{build}(x, \{a \mid \sigma[i](p)(a)\})$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) := \mathbf{not}(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) :=$ $\mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\ominus \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\, \varphi) := \mathbf{exists}(\langle x_1, \ldots, x_k \rangle, \mathsf{now}(\varphi))$.
5. If $\mathsf{now}(\eta) =$ BDD($\bot$) then report a violation, otherwise goto step 2.

Example 3. Consider a satellite with several radios on board, over which telemetry data are transmitted to ground (Earth). Consider the property that asserts that when $\mathit{telem}(x, d)$ appears in a state, denoting that telemetry data d are transmitted over radio x, then radio x has been opened in the past for communication with some frequency f, and not closed since. This property can be stated as follows in PFLTL:

$$\forall x.(\exists d(\mathit{telem}(x,d))) \rightarrow (\exists f(\neg \mathit{close}(x)\ \mathcal{S}\ \mathit{open}(x,f))) \quad (8)$$

As previously explained, the subformulas of this formula are evaluated bottom up, with the subformulas of a formula being evaluated before the formula itself. Figure 2 illustrates how this formula is broken down into enumerated subformluas by the DEJAVU tool, effectively an Abstract Syntax Tree (AST). For each event these subformulas are then evaluated in the following order: 8, 7, 6, 5, 4, 3, 2, 1, 0. We shall illustrate the BDDs generated for subformula 5: $\neg \mathit{close}(x)\ \mathcal{S}\ \mathit{open}(x, f)$ during evaluation of a trace with the following prefix: $\{\mathit{open}(A, 145)\}, \{\mathit{open}(B, 440)\}, \{\mathit{telem}(A, 42)\}$.

After the first event $\mathit{open}(A, 145)$ the data values A respectively 145 are first mapped to enumerations. Each variable x and f is enumerated individually and hence the same enumeration can in principle be used for representing a value for x as well as for f. In this case we choose the enumerations suggested by the implementation of our algorithm. Using three bits per variable, binary 110 (Natural number 6) is chosen as enumeration for x as well as for f. If we name the BDD variables for x as x_1, x_2, and x_3 with x_1 being the least significant bit, and name the bits for f as f_1, f_2, and f_3 with f_1 being the least significant bit, then the BDD representing the assignment $[x \mapsto A, f \mapsto 145]$ becomes the one shown in Figure 3.

Each node in the BDD represents one of the Boolean variables (bits) x_1, x_2, x_3, f_1, f_2, or f_3. The leaf nodes 1 and

[6] The number of bits needed depends on how many different values can be observed in a trace, which of course cannot be determined up front. Note, however, that with k bits we can store 2^k enumerations, which for any k corresponds to an exponentially larger number of enumerations. Note also, that it is possible to extend the number of bits used on the fly. Note finally, that we have implemented a garbage collection procedure which re-collects enumerations no longer needed.

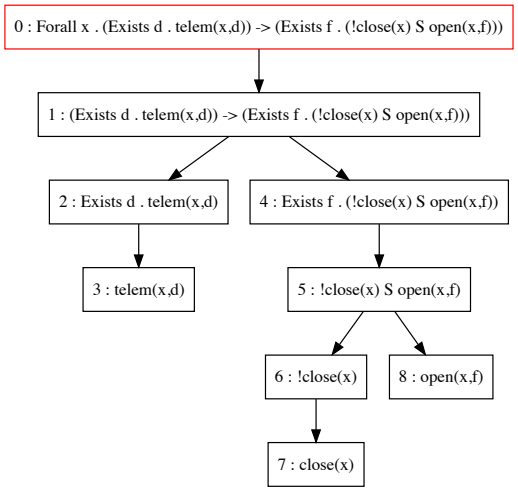


Fig. 2: AST for first-order telemetry property.

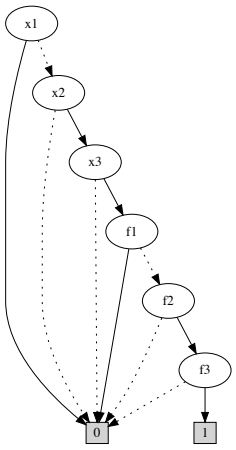


Fig. 3: BDD denoted by subformula $\neg close(x)\ \mathcal{S}\ open(x,f)$ after first event. It denotes the Boolean variable assignment $x_3x_2x_1f_3f_2f_1 = 110110$.

0 represent true respectively false. Note that least significant bits, x_1 and f_1, appear towards the top of the BDD, and the most significant bits appear last, just before a leaf. Recall that in BDDs, a bit may not appear in a path, in the case that the truth value for the path will be the same independent of this bit. The BDD defines the valid assignments of 0 and 1 values to the six Boolean variables as follows. From each non-leaf node, a dotted-line arrow represents the Boolean value 0 and a thick-line node represents the Boolean value 1. A path from the top node x_1 to the leaf-node 1 represents one valid assignment. In this case the assignment: $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $f_1 = 0$, $f_2 = 1$, and $f_3 = 1$, corresponding to the bit vector $x_3x_2x_1f_3f_2f_1 = 110110$. This corresponds to the enumeration 110 being assigned to x as well as to f.

At the second event $open(B, 440)$ we again assign enumerations to x and f representing the values B and 440, this time the binary enumeration 101 (Natural number 5) for each. At this point the subformula $\neg close(x)\ \mathcal{S}\ open(x,f)$ denotes the BDD shown in Figure 4, which represents the set of assignments: $\{[x \mapsto A, f \mapsto 145], [x \mapsto B, f \mapsto 440]\}$. In this BDD the leftmost path from x_1 to leaf-node 1 is the path from Figure 3. The new path is the rightmost path representing the bit pattern: $x_3x_2x_1f_3f_2f_1 = 101101$. Stated differently, the set of assignments $\{[x \mapsto A, f \mapsto 145], [x \mapsto B, f \mapsto 440]\}$ is now represented by a BDD corresponding to the Boolean expression: $(\neg x_1 \wedge x_2 \wedge x_3 \wedge \neg f_1 \wedge f_2 \wedge f_3) \vee (x_1 \wedge \neg x_2 \wedge x_3 \wedge f_1 \wedge \neg f_2 \wedge f_3)$.

When processing the third event $telem(A, 42)$, also here we need to assign an enumerations to A and 42. It turns out, however, that A has already been assigned the enumeration 110 in processing of the first event, so we only need to assign a new enumeration for 42. Note, however, that the subformula $telem(x,d)$ of the original formula only stores a BDD representing this single assignment in the current step. It is forgotten when we move on to the next event since that subformula does not occur under a temporal past time operator.

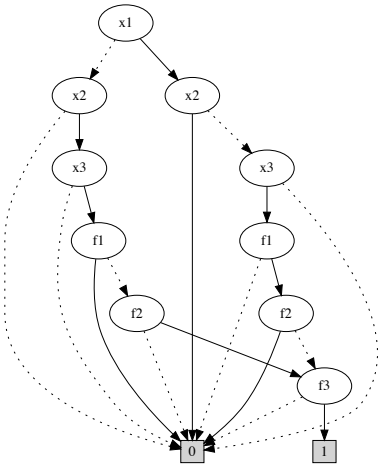


Fig. 4: BDD denoted by subformula $\neg close(x)\ \mathcal{S}\ open(x,f)$ after second event. It denotes the Boolean variable assignments $x_3x_2x_1f_3f_2f_1 = 110110$ and $x_3x_2x_1f_3f_2f_1 = 101101$.

RV algorithm for RPFLTL We extend now the algorithm to capture RPFLTL. The auxiliary relations r_j extend the model, and we need to keep BDDs representing $\mathsf{now}(r_j)$ and $\mathsf{pre}(r_j)$ for each relation r_j. We also need to calculate the subformulas φ_i that appear in a specification. One subtle point is that the auxiliary relations r_j may be defined in a

rule with respect to a variable x_j as in $r_j(x_j) := \varphi_j(x_j)$ (this can be generalized to any number of variables), but r_j can be used as a subformula with other parameters in other rules or in the statement e.g., as $r_j(y)$. This can be resolved by a BDD renaming function **rename**$(r_j(x_j), y)$ where the BDD bits of x_j are renamed to the BDD bits of y. We then add the following updates to step 4 of the above algorithm.

For each rule $r_j(x_j) := \varphi_j(x_j)$:
  calculate $\mathsf{now}(\varphi_j)$;
  $\mathsf{now}(r_j) := \mathsf{now}(\varphi_j)$;
  $\mathsf{now}(r_j(y)) :=$ **rename**$(r_j(x_j), y)$;
  $\mathsf{now}(r_j(a)) :=$ if $\mathsf{now}(r_j)(a)$ then BDD$(\top)$ else BDD$(\bot)$

As in the propositional case, the evaluation order cannot be simply top down or bottom up, since relations can appear both on the left and the right of a definition such as $r(x) := p(x) \vee \ominus r(x)$; we need to use the *mixed evaluation order*, described in Section 3.

Complexity. BDDs were first introduced to model checking [8] since they can often (but not always) allow a very compact representation of states. In our context, each BDD in pre or now represents a relation with k parameters, which summarizes the value of a subformula of the checked PFLTL or RPFLTL property with k Boolean variables over the prefix observed so far. Hence, it can grow up to a size that is polynomial in the number of values appearing in the prefix, and exponential in k (with k being typically very small). However, the combination of BDDs and Boolean enumeration is in particular efficient, since collections of adjacent Boolean enumerations tend to compact well.

We return now to the proof of Theorem 2. For that, we use the following Lemma.

Lemma 6 *Let η be some RPFLTL formula, and σ, σ' be two sequences of states (or events) such that the summary of η after σ is the same as after σ'. Then for each sequence of states (events) ρ, the summary of η after $\sigma\rho$ is the same as after $\sigma'\rho$.*

Proof. By a simple induction on the length of ρ. □

Proof of Theorem 2. The proof of this theorem includes encoding of a property that observes sets of data elements, where elements a, appears separately, i.e., one per state (or event), as $v(a)$, in between states where r appears. The domain of data elements is unbounded. The set of a-values observed in between two consecutive r's is called a *data set*. The property asserts that no data set appears twice. This property can be expressed in QPFLTL. We use two auxiliary 0-ary (i.e., constant) relations p and q to mark two different data sets that appeared in the past. The property that expresses that there exists in the past a data set where each state satisfies p is as follows: $\exists p\,(\neg p \mathcal{S}((r \wedge \neg p) \wedge \ominus((p \wedge \neg r)\mathcal{S}(r \wedge H\neg p))))$ The property for q is obtained by replacing p by q. We also need to assert that $H\neg(p \wedge q)$ (p and q do not happen together), and $(\forall x\, Pv(x) \wedge p) \leftrightarrow P(v(x) \wedge q)$, which expresses that the data sets that is annotated by p and by q have the same elements.

We use a combinatorial argument to show by contradiction that one cannot express this property using any RPFLTL formula φ. Let D of size m be a set of m values, and consider all the finite sequences consisting each of datasets without repetition. There are 2^m possible data sets, and 2^{2^m} sets of datasets. Thus $o(2^{2^m})$ such sequences (note that this is a lower bound, in fact, a much larger number of sequences exist, since the different datasets can be permuted in any order). A summary for the RPFLTL algorithm is bounded by $O((m+1)^N)$, where N is the maximal number of parameters in a relations in the property. We use $m+1$ rather than m, since there is one representation for all the values not used (corresponding to the enumeration $11\ldots11$). But in order to distinguish between $o(2^{2^m})$ possibilities of sets of datasets, we need memory of size $o(2^m)$. This means that with large enough m, each RPFLTL formula φ over the models of this property can have two sequences with the same summary, where one of them has some data set that the other one does not. We now extend these two prefixes with that distinguishing dataset. But according to Lemma 6, both of these extended sequences will satisfy or both will falsify φ, a contradiction to the assumption that φ satisfies the required property. □

6 Implementation

DEJAVU is implemented in the SCALA programming language. DEJAVU takes as input a specification file containing one or more properties, and synthesizes the monitor as a self-contained SCALA program. This program takes as input the trace file and analyzes it. The tool uses the JavaBDD library for BDD manipulations [24].

Example properties. Figure 5 shows four properties in the input ASCII format of the tool, the first three of which are related to the examples in Section 4, which are not expressible in (P)FLTL. That is, these properties are not expressible in the original first-order logic of DEJAVU presented in [18]. The last property illustrates the use of rules to perform conceptual abstraction. The ASCII version of the logic uses @ for $\ominus$, | for $\vee$, & for $\wedge$, and ! for $\neg$. The first property telemetry1 is a variation of the radio-telemetry property (8) in Example 3. It uses a rule, as in (7), to express a first-order version of Wolper's example [32], that all the states with even indexes of a sequence satisfy a property. In this case we consider a radio on board a spacecraft, which communicates over different channels (quantified over in the formula) that can be turned on and off with a toggle(x); they are initially off. Telemetry can only be sent to ground over a channel x with the telem(x) event when radio channel x is toggled on.

```
prop telemetry1 : Forall x .
  closed(x) → !telem(x) where
      closed(x) := toggle(x) ↔ @!closed(x)

prop telemetry2 : Forall x .
  closed(x) → !telem(x) where
      closed(x) :=
          (!@true & !toggle(x)) |
          (@closed(x) & !toggle(x)) |
          (@open(x) & toggle(x)),
      open(x) :=
          (@open(x) & !toggle(x)) |
          (@closed(x) & toggle(x))

prop spawning : Forall x . Forall y . Forall d .
  report(y,x,d) → spawned(x,y) where
      spawned(x,y) :=
          @spawned(x,y) |
          spawn(x,y) |
          Exists z . (@spawned(x,z) & spawn(z,y))

prop commands : Forall c .
  dispatch(c) → ! already_dispatched(c) where
      already_dispatched(c) := @ [dispatch(c) , complete(c)),
      dispatch(c) := Exists t . CMD_DISPATCH(c,t),
      complete(c) := Exists t . CMD_COMPLETE(c,t)
```

Fig. 5: Properties stated in DEJAVU's logic.

The second property, telemetry2, expresses the same property as telemetry1, but in this case using two rules, reflecting how we would model this using a state machine with two states for each channel x: closed(x) and open(x). The rule closed(x) is defined as a disjunction between three alternatives. The first states that this predicate is true if we are in the initial state (the only state where **@true** is false), and there is no toggle(x) event. The next alternative states that closed(x) was true in the previous state and there is no toggle(x) event. The third alternative states that in the previous state we were in the open(x) state and we observe a toggle(x) event. Similarly for the open(x) rule.

The third property, spawning, expresses a property about threads being spawned in an operating system. We want to ensure that when a thread y reports some data d back to another thread x, then thread y has been spawned by thread x either directly, or transitively via a sequence of spawn events. The events are spawn(x,y) (thread x spawns thread y) and report(y,x,d) (thread y reports data d back to thread x). For this we need to compute a transitive closure of spawning relationships, here expressed with the rule spawned(x,y).

The fourth property, commands, concerns a realistic log from the Mars rover Curiosity [28]. The log consists of events (here renamed) CMD_DISPATCH(c,t) and CMD_COMPLETE(c,t), representing the dispatch and subsequent completion of a command c at time t. The property to be verified is that a command, once dispatched, is not dispatched again before completed. Rules are used to break down the formula to conceptually simpler pieces. This property can be expressed without the use of rules, but the result will be harder to comprehend.

Example Execution. We shall briefly illustrate how property telemetry1 is evaluated on a trace. Figure 6 (generated by DEJAVU) shows the structure (AST) of the formula, as it is represented internally by DEJAVU. The reader may compare to the AST in Figure 2 for the first-order property (8). The AST (stored in now and pre) contains all nodes needed for evaluation of the property. Two new kinds of nodes occur, compared to Figure 2, namely arrow shaped boxes (nodes number 2 and 9) representing rule calls, and a tagged box (node number 5) representing the body of the rule. A dotted arrow from a rule call (nodes 2 and 9) leads to the body (node 5) of the rule called. This has the meaning that the "calling" (arrow-shaped) node will denote the same BDD value as the "called" node.

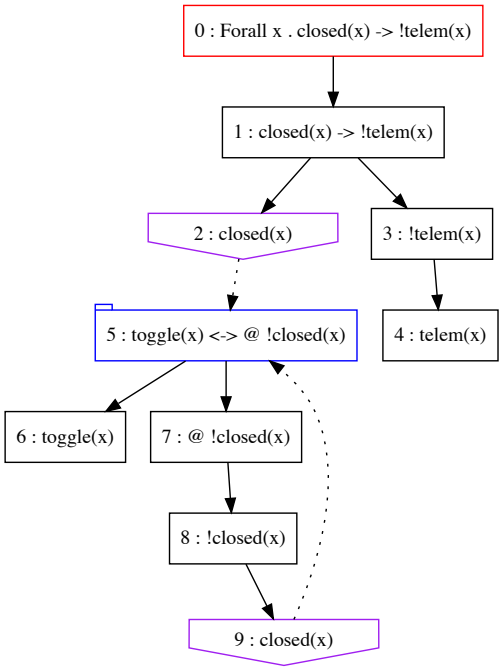


Fig. 6: AST for telemetry1 property, illustrating a rule.

The property-specific part of the synthesized monitor[7] is shown in Figure 7. This function is called for each new event. The function operates on the two arrays holding BDDs, indexed by subformula indexes: pre for the previ-

[7] An additional 900+ lines of mostly property independent boilerplate code is generated.

ous state and now for the current state. For each observed event, the function evaluate() computes the now array, evaluating any subformula before any formula containing it, and returns *true* (the property is satisfied in this position of the trace) iff now(0) is not BDD($\bot$), which effectively means it is BDD($\top$) since the top-level formula contains no free variables. The evaluation uses *mixed evaluation order* according to steps (a)-(d) below. One can compare it to the bottom-up evaluation order, as in the PFLTL case, in Figure 2.

(a) Evaluate subformulas of the rule body, which are *not* within the scope of a $\ominus$ operator (nodes 7, 6). Observe that now($\ominus\gamma$) is set to pre(γ).
(b) Evaluate the top level rule body (node 5).
(c) Evaluate each subformula that appears in the rule body *within* the scope of a $\ominus$ operator (nodes 9, 8).
(d) Finally, evaluate the main formula (nodes 4, 3, 2, 1, 0).

At composite subformula nodes, BDD operators are applied. For example for subformula 1, the new value is now(2).not().or(now(3)), which is the interpretation of the formula closed(x) $\rightarrow$!telem(x) corresponding to the Boolean equivalence $(p \rightarrow q) \equiv (\neg p \vee q)$.

```
override def evaluate(): Boolean = {
  // a. formulas in rule rhs not below @:
  now(7) = pre(8)
  now(6) = build("toggle")(V("x"))

  // b. rule body
  now(5) = now(6).biimp(now(7))

  // c. formulas in rule rhs below @:
  now(9) = now(5)
  now(8) = now(9).not()

  // d. main formula:
  now(4) = build("telem")(V("x"))
  now(3) = now(4).not()
  now(2) = now(5)
  now(1) = now(2).not().or(now(3))
  now(0) = now(1).forAll(var_x.quantvar)

  // Calculate result and move now to pre:
  val error = now(0).isZero
  tmp = now; now = pre; pre = tmp
  !error
}
```

Fig. 7: Monitor evaluation function for property telemetry1.

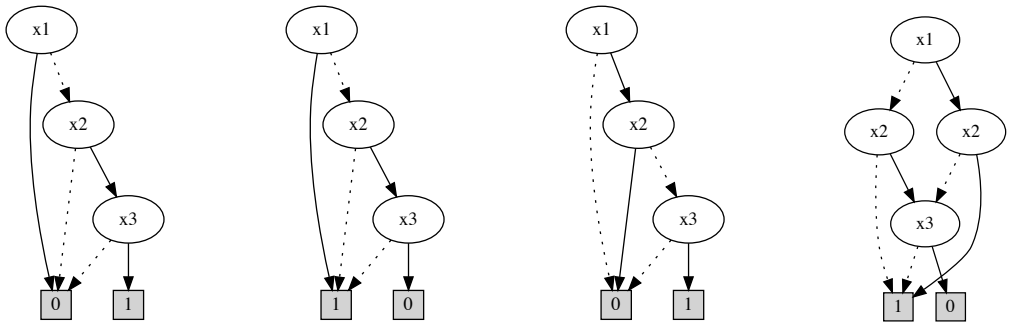
As an example of how a trace is processed consider the following simple (correct) trace containing three events: $\{toggle(L)\}, \{toggle(H)\}, \{telem(L)\}$. This represents turning on low (L) and high (H) frequency channels (recall that they are initially off), and sending telemetry on channel L. Upon encounter of the first event $toggle(L)$, the value L is assigned the binary enumeration 110 (the Natural number 6) in node 6, and is represented as the BDD in Figure 8a. We represent each enumeration for the variable x with three Boolean variables (bits) x_1, x_2, and x_3, where x_1 is the least significant bit. Hence $x_3x_2x_1 = 110$ for the first event. The BDD represents all assignments to the Boolean variables x_1, x_2, and x_3 that lead to the leaf-node 1 (true). (Recall that in our implementation, the top node corresponds to the least significant bit, hence BDD bits appear in a path from the root to the leaf node 1 in the order 011).

Figure 8b shows the BDD assigned to node 5. This is the BDD corresponding to the BDD from node 6 negated, reflecting that all radios are closed, *except* for (the enumeration corresponding to) L. Note that the path 011 leads to leaf-node 0 in contrast to Figure 8a (negating a BDD is achieved by just flipping the 0 and 1 leaf-nodes). To understand this result consider that node 7 is BDD($\bot$) since initially @!closed(x) (node 7) is BDD($\bot$) in the first state, and consider that for a given Boolean expression $\mathcal{B}$, it holds that $(\mathcal{B} \leftrightarrow false) \equiv \neg\mathcal{B}$.

Upon encountering the second event $toggle(H)$, the value H is assigned the binary enumeration 101 (the Natural number 5). Figure 8c shows the BBD assigned to node 6 for the second value H, corresponding to the enumeration $x_3x_2x_1 = 101$. Figure 8d shows the BDD representing the rule closed(x) in node 5 after these two events. Since channels L and H, corresponding to enumerations $x_3x_2x_1 = 110$ and $x_3x_2x_1 = 101$, are now open, this BDD represents all the *other* enumerations. That is, in this BDD all enumerations, *except* $x_3x_2x_1 = 110$ and $x_3x_2x_1 = 101$, lead to leaf-node 1, whereas the two mentioned enumerations themselves both lead to leaf-node 0.

Upon the arrival of the third event $telem(L)$, the previously generated enumeration $x_3x_2x_1 = 110$ for L is looked up and assigned to node 4. Its negation, equivalent to the BDD in Figure 8b, is stored in node 3. Node 1 is computed as now(2).not().or(now(3)). Since node 2 represents all enumerations different from $x_3x_2x_1 = 110$ and $x_3x_2x_1 = 101$, now(2).not() represents exactly those enumerations. Node 1 therefore is the union of those enumerations and all enumerations different from $x_3x_2x_1 = 110$, effectively yielding all enumerations, represented as BDD($\top$), which then also becomes the BDD of node 0, and the property is satisfied.

Evaluation. In [18, 19] we evaluated DEJAVU without the rule extension against the MONPOLY tool [4], which supports a logic close to DEJAVU's. In [17] we evaluated DEJAVU's garbage collection capability. In this section we evaluate the rule extension for the properties in Figure 5 on a collection of traces. Table 1 shows the analysis time (excluding time to compile the generated monitor) and maximal memory usage in MB (megabytes) for different traces (format is 'trace length : time / memory').



(a) Node 6 after 1st ev. (b) Node 5 after 1st ev. (c) Node 6 after 2nd ev. (d) Node 5 after 2nd ev.

Fig. 8: Selected BDDs from trace evaluation.

Property	Trace 1	Trace 2	Trace 3
telemetry1	1,200,001 : 2.6s / 194 MB	5,200,001 : 5.9s / 210 MB	10,200,001 : 10.7s / 239 MB
telemetry2	1,200,001 : 3.8s / 225 MB	5,200,001 : 8.7s / 218 MB	10,200,001 : 16.6s / 214 MB
spawning	9,899 : 29.5s / 737 MB	19,999 : 117.3s / 1,153 MB	39,799 : 512.5s / 3,513 MB
commands	49,999 : 1.5s / 169 MB	N/A	N/A

Table 1: Evaluation - trace lengths, analysis time in seconds, and maximal memory use.

The processing time is generally very reasonable for very large traces. However, the spawning property requires considerably larger processing time and memory compared to the other properties since more data (the transitive closure) has to be computed and stored. The evaluation was performed on a Mac laptop, with the Mac OS X 10.10.5 operating system, on a 2.8 GHz Intel Core i7 with 16 GB of memory.

7 Conclusions

Propositional linear temporal logic (LTL) and automata are two common specification formalisms for software and hardware systems. While temporal logic has a more declarative flavor, automata are more operational, describing how the specified system progresses.

Several extensions of propositional LTL have been proposed by others to increase its expressive power to that of related automata formalisms. We proposed here a simple extension for propositional LTL, which adds auxiliary propositions that summarize the prefix of the execution based on rules written using past time temporal formulas. This extension puts the logic, conceptually, in between propositional LTL and automata, as the additional variables can be seen as representing the state of an automaton that is synchronized with the temporal property. It is shown to have the same expressive power as Büchi automata. It is in particular appealing for runtime verification of past temporal properties, which already are based on summarizing the value of subformulas over observed prefixes. Hence extending existing RV algorithms accordingly is simple and requires no additional complexity.

We demonstrated that first-order linear temporal logic (FLTL), which can be used to express properties about systems with data, also has expressiveness deficiencies, and similarly extended it with rules that define *relations* that summarize prefixes of the execution. We proved that for the first-order case, unlike the propositional case, this extension is not identical to the addition of dynamic (i.e., state dependent) quantification.

We presented a monitoring algorithm for propositional past time temporal logic with rules, extending a classical algorithm, and similarly presented an algorithm for first-order past temporal logic with rules. Finally we described the implementation of this extension in the DEJAVU tool and provided experimental results. The code and many more examples appear at [10]. Future work includes making further comparisons between the different version of first order LTL logics and to other formalisms, in particularly automata based. We intend to study further extensions, exploring the space between logic and programming.

Acknowledgements We would like to thank Dogan Ulus for his contributions to the earlier stages of this project. We would also like to thank Kim Guldtrand Larsen, Ioannis Filippidis, Armin Bierre, and Jaco van de Pol for sharing their BDD expertise with us.

References

1. B. Alpern, F. B. Schneider, Recognizing safety and liveness. Distributed Computing 2(3): 117-126, 1987.
2. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna: LOLA: Runtime monitoring of synchronous systems, TIME 2005, 166-174.
3. E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, G. Reger, An introduction to runtime verification, lectures on runtime verification - introductory and advanced topics, LNCS Volume 10457, Springer, 1-23, 2018.
4. D. A. Basin, F. Klaedtke, S. Marinovic, E.n Zalinescu, Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design 46(3): 262-285, 2015.
5. A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly?, RV'07, LNCS Volume 4839, Springer, 126-138, 2007.
6. J. Bohn, W. Damm, O. Grumberg, H. Hungar, K. Laster, First-order CTL model checking. FSTTCS 1998: 283-294.
7. R. E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, ACM Computing Survety 24(3), 293-318, 1992.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic model checking: 10^{20} states and beyond, Information and Computation 98(2): 142-170 (1992).
9. J. Chomicki, Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2): 149-186, 1995.
10. DejaVu, https://github.com/havelund/dejavu.
11. H.-D. Ebbinghaus, J. Flum, W. Thomas, Mathematical logic. Undergraduate texts in mathematics, Springer 1984.
12. C. Colombo, G. J. Pace, and G. Schneider. LARVA - Safer monitoring of real-time Java programs. In 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09), pages 33-37, Hanoi, Vietnam, 23-27 November 2009. IEEE Computer Society.
13. Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime? STTT 14(3), 349-382, 2012.
14. H. Frenkel, O. Grumberg, S. Sheinvald, An automata-theoretic approach to modeling systems and specifications over infinite data. NFM 2017, 1-18.
15. S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
16. K. Havelund, Rule-based runtime verification revisited, STTT 17(2), 143-170, 2015.
17. K. Havelund, D. Peled, Efficient runtime verification of first-order temporal properties. SPIN 2018, Malaga, Spain, 26-47.
18. K. Havelund, D. A. Peled, D. Ulus, First-order temporal logic monitoring with BDDs, FMCAD 2017, 116-123.
19. K. Havelund, D. A. Peled, D. Ulus, First-order temporal logic monitoring with BDDs, Formal Methods in System Design: 1-21, 2019.
20. K. Havelund, G. Reger, D. Thoma, and E. Zălinescu, Monitoring events that carry data, lectures on runtime verification - introductory and advanced topics, LNCS Volume 10457, Springer, 61-102, 2018.
21. K. Havelund, G. Rosu, Synthesizing monitors for safety properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
22. L. Hella, L. Libkin, J. Nurmonen, L. Wong, Logics with aggregate operators. J. ACM 48(4): 880-907, 2001.
23. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850TM-2010, 2010.
24. JavaBDD, http://javabdd.sourceforge.net.
25. O. Kupferman, M. Y. Vardi, Model checking of safety properties. Formal Methods in System Design 19(3): 291-314, 2001.
26. Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, 1992.
27. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, STTT, Springer, 249-289, 2011.
28. Mars Science Laboratory (MSL) mission website. http://mars.jpl.nasa.gov/msl.
29. G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at runtime with QEA, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), Springer, 2015.
30. A. P. Sistla, Theoretical Issues in the Design and Analysis of Distributed Systems, Ph.D Thesis, Harvard University, 1983.
31. W. Thomas, Automata on infinite objects, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, 133-192, 1990.
32. P. Wolper, Temporal logic can be more expressive, Information and Control 56(1/2): 72-99, 1983.
33. P. Wolper, M. Y. Vardi, A. P. Sistla: Reasoning about infinite computation paths (Extended Abstract). FOCS 1983, 185-194.