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

An Extension of First-Order LTL with Rules with 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 [45] 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, by and large, have been adopted for formal methods. We present an extension of propositional LTL with rules, that is as expressive as these aforementioned extensions. We then show a similar deficiency in the expressiveness of *first-order* LTL and present an extension of it with rules, which parallels the propositional version. In our work on runtime verification we focus on execution traces which consist of events that carry data, where a first-order version of LTL is needed, and in particular on past time versions of first-order LTL. In previous work we provided an algorithm for past time first-order LTL that uses BDDs to represent relations over data elements, and implemented it as a tool called DEJAVU. In this paper we propose a monitoring algorithm for the extension of past time first-order LTL with rules. This is implemented as an extension of DEJAVU, and experimental results are provided.

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) [6, 29] 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 on specification-based runtime verification, where an execution trace is verified against a formal specification, and in particular where the specification logic is a variant of past time first-order Linear Temporal Logic (LTL). This paper first introduces new theoretical results related to full LTL, including future time as well as past time operators. For runtime verification, it then narrows in on the past time first-order case and presents extensions to existing algorithms. One of these extensions is then implemented in the DEJAVU tool.

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 [44], all having the same expressive power. In fact, model checking [14] of an LTL property is usually performed by first translating the property into a Büchi automaton [22]. The automata formalisms are more expressive than LTL; a classical example by Wolper [45] shows that it is not possible to express in LTL the property that *every even state in a sequence satisfies some proposition p*. We will use this example several times throughout the paper to demonstrate a weakness of different versions of temporal logic, and suggest extensions to these versions, where this weakness is repaired.

Several extensions of LTL were proposed, in order to achieve the same expressive power as Büchi automata: Wolper's ETL [45, 46] uses right-linear grammars, Sistla's QLTL extends LTL with dynamic (i.e., state-dependent, second-order) *quantification over proposi-*

tions [43], and the PSL standard [39] extends LTL with regular expressions. However, these and other extensions have not been largely used for RV.

We present here an alternative extension of propositional LTL with *rules*, named RLTL, which is suited for RV. 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 conforms easily and naturally with existing RV algorithms that use incremental summaries of prefixes, e.g., the classical algorithm [30] for past time LTL (denoted here PLTL), 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 expressive-equivalent to QLTL and its restriction to past properties, RPLTL, is expressive-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 to here as FLTL. We show that the weakness of propositional LTL, as demonstrated by Wolper's property, can be lifted to a related property that is inexpressible in the first-order FLTL. Further, we show that this latter logic also lacks the power to express the transitive closure of temporal relations over events. We then introduce two alternative ways of extending the expressive power of FLTL, corresponding, respectively, to the propositional logics QLTL and RLTL. The first alternative 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 Wolper's property, relativized to the first-order case, and also the transitive closure of temporal relations. We show that, in contrast to the propositional case, where the extension of RLTL of LTL with rules is as expressive as the extension QLTL with dynamic quantification, in the first-order case, the extension RFLTL with rules is less expressive than the extension QFLTL with dynamic quantification.

In our work on runtime verification, we focus on the *past time* versions of LTL, which express *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 past logics also enjoy elegant RV algorithms, based on the ability to compute summaries of the observed prefixes [27, 30], as opposed to future temporal logics [8].

We extend the algorithm in [30] for propositional past time LTL with rules to the logic RPLTL (the past part of RLTL). The structure of the rules blends well with the RV algorithm. In fact, the definition of the rules are inspired by the RV algorithm, in particular the summary, and consequently the extension of the algorithm is simple. Our main result, the RV algorithm for past time first-order LTL extended with rules, RPFLTL (the past part of RFLTL), naturally extends the RV algorithm for past time first-order LTL, PFLTL, presented in [27].

We describe the corresponding extension of the RV tool DEJAVU [26, 27, 28], available at [16], which monitors past time first-order LTL (PFLTL) to include rules (i.e., to RPFLTL). The DEJAVU tool allows runtime verification of past time first-order temporal logic over infinite domains (e.g., 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 use of BDDs that is different from the classical model checking representation of sets of Boolean states; e.g., in [10], BDDs are used to represent sets of program locations, and sets of data elements are represented symbolically as formulas.

Related work. Several linear temporal logics for RV, supporting data parameterization, have been developed over the past two decades. Our work differs by augmenting first-order LTL with rules, and by representing data with BDDs. An RV algorithm for the first-order linear temporal logic MFOTL was presented in [8], and implemented in the MONPOLY tool, based on two alternative approaches. In the first one, negation can appear unrestricted within the temporal formula, and relations are represented as *regular sets* and, subsequently, automata. In the second one, negations are restricted and relations are represented explicitly as sets of tuples and are subjected to database operators (e.g., join). MFOTL also supports aggregation operators (e.g., sum and average) [7], increasing the expressiveness of the logic.

MOP [36] offers several data parametric specification formalisms as separate plugins, including past and future temporal logics, regular expressions, state machines, grammars, etc. The logics, however, are separated in the sense that any property is expressed in one of the logics. The parameterization is based on slicing and offers a somewhat limited expressiveness. The QEA system [40] is an automaton-based approach, which improves the expressiveness of the slicing approach. A system for monitoring first-order future time LTL using an SMT solver is described in [15]. Formalisms based on formula rewriting are used in BEEPBEEP [23], which is based on future time linear temporal logic, and DAUT [24], an internal Scala DSL for programming monitors in a combination of rule systems and state machines. The MESA system [42] is an actor-based RV system utilizing concurrency to improve monitoring performance, using DAUT as its temporal logic. STL (Signal Temporal Logic) [34] is an extension of metric temporal logic

with numerical predicates, allowing to express mixed-signal properties over real valued variables as well as Boolean valued variables. R2U2 [21] offers the metric temporal logic MTL, which adds time bounds to LTL's temporal operators. R2U2 is implemented on dedicated FPGA hardware, enabling it to monitor the health status of e.g., Unmanned Aerial Systems (UAS). Although the logic can be classified as propositional, suitable for hard real-time constraints, it allows propositions to be Boolean expressions over monitored variables.

Temporal logics have been defined using recursion. EAGLE [4] was one of the first systems to provide a temporal logic supporting data parameterized events. It implements a recursive calculus with past and future time operators. DETECTER [3] implements a future time data parametric Hennessy-Milner logic with recursion for monitoring ERLANG programs. Previous work includes purely rule-based systems, e.g., RULER [5], and later LOGFIRE [25], which offers an internal SCALA DSL, the implementation of which is based on the classical Rete algorithm [19] for rule systems, augmented with handling of events for RV. These systems, however, do not focus on temporal logics, although limited temporal logics were defined and mapped into these rule systems.

A different branch of formalisms include those supported in stream-based systems, such as LOLA [2] and COPILOT [38]. The COPILOT specification language is an embedded HASKELL DSL from which monitors in C are generated for monitoring hard real-time reactive systems. It supports a past time linear temporal logic and a bounded future time temporal logic, both mapped into stream expressions. It supports data parameterization, which, however, is bounded due to the real-time constraints requiring statically bounded execution time and memory usage. One must, e.g., put a bound on for how long a monitor should remember the value of a variable. The stream processing makes COPILOT more expressive than LTL.

Paper outline. The structure of the paper reflects our stepwise 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. Section 2 describes a limitation of propositional LTL pointed out by Wolper, and proposes a rule-based extension of propositional LTL. Section 3 proposes an RV algorithm for monitoring the past time subset of the rule-based extension of propositional LTL. Section 4 introduces first-order LTL, and shows a similar limitation of its expressiveness. We extend the logic in two directions, one with dynamic quantification and one with rules. Section 5 proposes an RV algorithm for monitoring the rule-based extension of past time first-order LTL. Section 6 presents the implementation of the latter algorithm in the DEJAVU tool and shows some experimental results. Finally Section 7 concludes the paper.

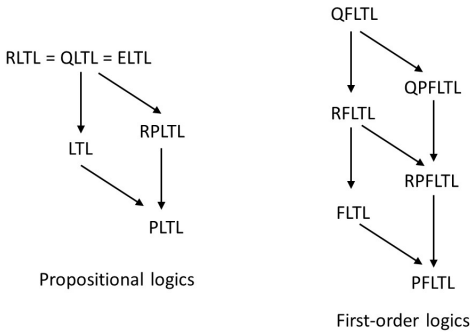


Fig. 1: Logics defined and discussed in this paper: P = Past, F = First-order, Q = Quantified, R = Rules. DEJAVU implements the logics RPFLTL, PFLTL, RPLTL and PLTL.

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, each arrow represents inclusion: an arrow from a logic X to a logic Y indicates that X is more expressive than Y when interpreted on infinite traces.

2 Propositional LTL

Linear temporal logic [35] 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 the temporal operators $\bigcirc$, $\mathcal{U}$, $\ominus$, $\mathcal{S}$ stand for *next-time*, *until*, *previous-time* and *since*, respectively. A model of an LTL formula is an infinite sequence of states, also referred to as a *trace*, of the form $\sigma = \sigma[1]\sigma[2], \sigma[3] \ldots$, where $\sigma[i] \subseteq P$ for $i \geq 1$. A state consists of the subset of the propositions, which *hold* in it. LTL's semantics is defined as follows, where $i \geq 1$ denotes a position in the trace:

- $(\sigma, i) \models \mathit{true}$.
- $(\sigma, i) \models p$ if $p \in \sigma[i]$.
- $(\sigma, i) \models (\varphi \wedge \psi)$ if $(\sigma, i) \models \varphi$ and $(\sigma, i) \models \psi$.
- $(\sigma, i) \models \neg\varphi$ if $(\sigma, i) \not\models \varphi$.
- $(\sigma, i) \models \bigcirc\varphi$ 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, $j < k \leq i$, $(\sigma, k) \models \varphi$.

We write $\sigma \models \varphi$ when $(\sigma, 1) \models \varphi$. We can use the following abbreviations: $false = \neg true$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $(\varphi \leftrightarrow \psi) = ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$, $\Diamond\varphi = (true\ \mathcal{U}\varphi)$ (eventually φ), $\Box\varphi = \neg\Diamond\neg\varphi$ (always in the future φ), $\ominus\varphi = (true\ \mathcal{S}\varphi)$ (previously φ) and $\boxminus\varphi = \neg\ominus\neg\varphi$ (always in the past φ).

The expressive power of different versions of propositional LTL is often compared with finite automata over infinite words (Büchi, Street, Muler, Parity) and to *monadic* first and second-order logics[1]. Accordingly, LTL is equivalent to monadic first-order logic, and counter-free[2] Büchi automata. For an overview of logic and automata see [44]. 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, $\boxminus$ and $\ominus$. The past time logic is 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 latter 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. A failure to satisfy a safety property in an execution sequence (a model) can be detected after observing a finite prefix.

Wolper [45] demonstrated that the expressive power of LTL is lacking using the following example property.

Wolper's property:
All the states with even indexes in a sequence satisfy some proposition p.

Note that this is *different* from stating that p alternates between *true* and *false* on consecutive states, which *can* be expressed in LTL.

We now present two extensions of LTL, each enabling the formulation of Wolper's property; first a known extension using quantification, and then our contribution using rules, better suited for RV. In both cases new auxiliary propositions can be defined to hold in selected execution sequence positions, and used to define the property. The difference between the two approaches concerns the manner in which we define the positions where the auxiliary propositions hold.

Extending LTL with dynamic quantification. Adding quantification over propositions, suggested by Sistla in [43], 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. Informally, the property says that there exists a set of execution trace positions (states) where q is true, such that φ is true. Wolper's property can e.g., be expressed as:

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

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

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 type 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 dynamic 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 to obtain the full expressiveness of QLTL [44]. Restricting QLTL to the past modalities, one obtains the logic QPLTL. Property (1) above is expressed in QPLTL. QPLTL has the same expressive power as regular expressions and finite automata.

Extending LTL with rules. We introduce another extension of LTL, which we call RLTL. As will be shown in Section 3, this extension is natural for runtime verification. Instead of using quantification as in QLTL, we define the truth of each auxiliary proposition in different sequence positions as a function of past observations. 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 observed execution sequence.

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_j is a PLTL property; 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 the time variables, while second order logic allows quantifying over the 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 $xy^n z \in L$.

We refer to ψ as the *statement* of η. The statement is the main property. Each $a_j := \varphi_j$ called a *rule* (in text, rules will be separated by commas). The rules are used to define the truth values of the auxiliary propositions that are used in the statement η: in each state, each auxiliary variable a_j has a truth value that is defined using the past property φ_j, based on the prefix of the sequence, up and including the current state. Wolper's property 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.

Definition 2 The semantics can be defined as an extension to the above LTL semantics (page 3), 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 $a_2 := a_1$.

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. Let η be the RLTL formula ψ **where** $a_j := \varphi_j : j \in \{1, \ldots, n\}$. Let σ be a model with states over B. Then there is a unique model σ' such that $\sigma'|_A = \sigma$: the proof is by induction on the length of prefixes of σ'. 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 sketch. 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 [44] and then construct an RLTL formula that holds for the accepting executions of this automaton. The rules of this formula encode the automata transitions, with each rule encoding the value of a proposition in the next state based on the values of the propositions in the previous state. 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 if 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.*

3 RV for Propositional Past Time LTL and its Extension

Runtime verification of temporal logic specifications is often restricted to concentrate on past time properties. When monitoring past time specifications, one can distinguish a violation after observing a finite prefix of an execution, i.e., in finite time. For an extended discussion of the issue of *monitorability*, see e.g., [9, 18]. The RV algorithm for PLTL, presented in [30], is based on the observation that the semantics of the past time formulas $\ominus\varphi$ and $(\varphi \mathcal{S} \psi)$ in the current state is defined in terms of the semantics of its subformula(s) in the previous state. To demonstrate this, we rewrite the semantic definition of the $\mathcal{S}$ operator in a form that is more directly 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.

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* $\mathsf{now}(\psi)$.

- $\mathsf{now}(true) := true$.
- $\mathsf{now}(p) := (p \in s)$
- $\mathsf{now}(\varphi \wedge \psi) := (\mathsf{now}(\varphi) \wedge \mathsf{now}(\psi))$.
- $\mathsf{now}(\neg\varphi) := \neg\mathsf{now}(\varphi)$.
- $\mathsf{now}(\varphi \mathcal{S} \psi) := (\mathsf{now}(\psi) \vee (\mathsf{now}(\varphi) \wedge \mathsf{pre}(\varphi \mathcal{S} \psi)))$.
- $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.

5. If $\mathsf{now}(\eta) = false$ then report a violation, otherwise goto step 2.

Runtime verification for RPLTL. For RPLTL, we need to add to the above algorithm the 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 of now 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 a *mixed evaluation order*, where 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 in 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[3] $D_1, D_2, \ldots$, 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 $domain(x)$. For example $[x \to 5, y \to \text{“abc”}]$ assigns the values 5 to x and the value “abc” to y.

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

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

Additional operators are defined as in the propositional logic. We also 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. The syntax suggests the use of monadic (i.e., arity 1) relations. This is only done in order to simplify the presentation: all the syntactic and semantic definitions of the first-order logics that we present here, and the specifications handled by the tool DEJAVU, can use relations with arbitrary (finite) arities.

Semantics. A first-order *model* σ is a sequence $\sigma = \sigma[1]\sigma[2]\ldots$, where for each $i \geq 1$, $\sigma[i]$ 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 $free(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. Let ε be the empty assignment (with no variables). Let $\gamma[x \mapsto a]$ be the overriding of γ with the binding $[x \mapsto a]$. $(\gamma, \sigma, i) \models \varphi$ is defined where γ is an assignment over a set of variables that *includes* $free(\varphi)$, and $i \geq 1$:

- $(\gamma, \sigma, i) \models true$.
- $(\gamma, \sigma, i) \models p(a)$ 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 $a \in domain(x)$ 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 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 property] The following is a generalization of Wolper's property (page 4) to the first-order case. Let p and q be unary relations. The property 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, $p(a)$

[3] Finite domains are handled with some minor changes, see [27].

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 $\exists x\varphi$ 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 by *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 $\forall x\varphi$ are replaced by conjunctions. This results in an LTL (i.e., propositional) 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 the fact that Wolper's property (page 4) is not expressible in LTL contradicts the assumption that such a formula exists.

Parametric automata, used as a specification formalism in [20, 29, 36, 40], *can* express this property, where for each value a there is a separate automaton that counts modulo 2 the number of times that $q(a)$ has occurred.

Example 2 [Transitive closure] Consider the following property: $report(y,x,d)$ appears (holds) in a state in a sequence, denoting that process y sends some data d to a process x, only if there has been a chain of process spawns $spawn(x,x_1)$, $spawn(x_1,x_2)$ ... $spawn(x_l,y)$ holding in past states of that sequence. i.,e., y is a descendent process of x. The required property involves the transitive closure of the relation *spawn*. But the transitive closure of a relation property cannot be expressed in FLTL. To see this, note that FLTL can be translated, in a way similar to the standard translation of *LTL*, into monadic first-order logic [44], 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, a classical theory of first-order logic[4] says that the transitive closure of *spawn* cannot be expressed in the first-order setting.

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 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 is motivated by the propositional extension of LTL (PLTL, respectively) to RLTL (RPLTL, respectively) that was shown in Section 2.

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 free variable[5] 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 to RFLTL by adding a definition of the following form, 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 3 *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 4 (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.*

[4] This theory is based on the compactness theory of first-order logic, see [17] Chap. 4.

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

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:

$$\begin{array}{l}\forall x\forall y\forall d\,(\mathit{report}(y,x,d) \rightarrow \mathit{spawned}(x,y))\\ \quad \textbf{where } \mathit{spawned}(x,y) = ((\ominus \mathit{spawned}(x,y) \vee \mathit{spawn}(x,y)) \vee\\ \qquad \exists z\,(\ominus \mathit{spawned}(x,z) \wedge \mathit{spawn}(z,y)))\end{array}$$

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

In runtime verification of FLTL, the input consists of a sequence of events $\sigma[1]\,\sigma[2]\ldots$. An event is a finite set of relations, each being a finite set of tuples of data values. A typical use of runtime verification restricts each event $\sigma[i]$ to contain a single relation containing a single tuple. We shall often just refer to the tuple (labelled with the relation name) as the event.

Set semantics. We provide an alternative *set semantics* for the logic RPFLTL, which is equivalent to the above definition. Set semantics is more directly related to the calculation of the summary values in the RV algorithm that will be presented below. Under set semantics, introduced in [27] 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 $\mathit{hide}(\Gamma,W)$ 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 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 $i \geq 1$.

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

The semantics of $(\varphi\ \mathcal{S}\ \psi)$ reflects the following 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 [27] 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 [11]. 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 [26].

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[6]. The function **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[7].

We use x_1, x_2, and x_3 to represent the enumerations, with x_1 being the least significant bit. Assume 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). Then, **lookup**$(x,a) = 000$, and **lookup**$(x,b) = 001$; **build**(x,C) is a BDD that represents the *set* of values $C = \{$"a","b"$\}$, and is equivalent to the Boolean function $(\neg x_2 \wedge \neg x_3)$, which returns 1 for 000 and 001.

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 **exists**$(x, B_\varphi) = \exists x_1 \ldots \exists x_k B_\varphi$ 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 the RV algorithm, We 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$(\bot)$.
   - $\mathsf{now}(p(x)) :=$ **build**$(x, \{a \mid \sigma[i](p)(a)\})$.
   - $\mathsf{now}((\varphi \wedge \psi)) :=$ **and**$(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) :=$ **not**$(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) :=$ **or**$(\mathsf{now}(\psi),$ **and**$(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\ominus \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\, \varphi) :=$ **exists**$(x, \mathsf{now}(\varphi))$.
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:

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

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 and 145 are first mapped to enumerations. Each variable x and f is enumerated separately and hence the same enumeration can in principle be used for representing a value for x as well as for f. We use the enumerations suggested by the implementation of our algorithm. In this case, this is the binary 110 (the Natural number 6) for both the enumeration for A as well as for 145. We name the BDD variables for x as x_1, x_2, and x_3 (recall that x_1 is the least significant bit), and the BDD variables for f as f_1, f_2, and f_3. Then the BDD representing the assignment $[x \mapsto A, f \mapsto 145]$ becomes the one shown in Figure 3.

[6] Other enumeration generation schemes are possible. Our implementation allows a garbage collection mechanism that can reuse enumerations that are no longer needed.

[7] With k bits, we can store 2^k enumerations. It is possible to extend the number of bits used on the fly.

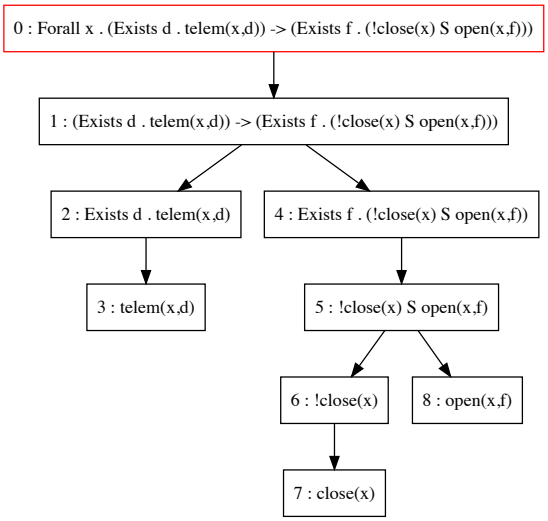


Fig. 2: Abstract Syntax Tree (AST) for the first-order telemetry property (8), showing its breakdown into subformulas. Nodes (subformulas) are numbered, with higher numbers lower in the tree, suggesting the bottom up evaluation from node 8 to node 0. The top node, corresponding to the complete formula, is colored red.

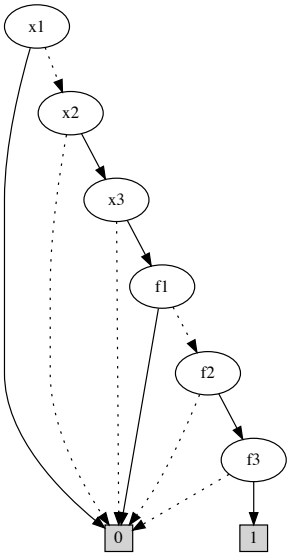


Fig. 3: The BDD for $\mathsf{now}(\neg close(x)\ \mathcal{S}\ open(x,f))$ after the first event, corresponding to the assignment $x_3x_2x_1f_3f_2f_1$ = 110110.

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 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 $f_3 = 1$, corresponds to the bit vector $x_3x_2x_1f_3f_2f_1$ = 110110; the enumeration 110 is 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 one of them. At this point the subformula $\mathsf{now}(\neg close(x)\ \mathcal{S}\ open(x,f))$ is 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.

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

For each rule $r(x) := \varphi(x)$:
  calculate $\mathsf{now}(\varphi)$;

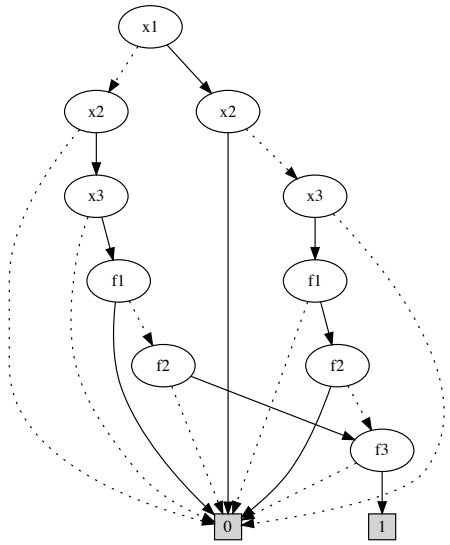


Fig. 4: The BDD for $\mathsf{now}(\neg close(x)\ \mathcal{S}\ open(x,f))$ after the second event, corresponding to the assignments $x_3x_2x_1f_3f_2f_1 = 110110$ and $x_3x_2x_1f_3f_2f_1 = 101101$.

$\mathsf{now}(r) := \mathsf{now}(\varphi)$;
$\mathsf{now}(r(y)) :=$ **rename**$(r(x), y)$;
$\mathsf{now}(r(a)) :=$ if $\mathsf{now}(r)(a)$ 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 [12] 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 can be quite compact, 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 5 *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 sketch of Theorem 2. The proof of this theorem includes encoding of a property that observes sets of data elements, where elements with value a, appears separately, i.e., one per state (or event), as $v(a)$, in between states where some event c appears (c is a constant, i.e., a relation with arity 0). The domain of data elements is unbounded. The set of a-values observed in between two consecutive c'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: $\exists p\,(\neg p \mathcal{S}((c \wedge \neg p) \wedge \ominus((p \wedge \neg c)\mathcal{S}(c \wedge H\neg p))))$. The property for q is obtained by replacing p by q. Our property asserts that $H\neg(p \wedge q)$ (p and q do not happen together), and $(\forall x P v(x) \wedge p) \leftrightarrow P(v(x) \wedge q)$, which expresses that the data sets 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 monitoring RV 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 5, 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. It 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 [32].

```
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.

Example properties. Figure 5 shows four properties in the input ASCII format of the tool. The first three of these 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 [27]. 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 $\neg$. Not shown in Figure 5 are the derived operators $\Diamond$ and $\Box$, which in ASCII format are written as **P** (Previously) and **H** (History) respectively. 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 property (see Example 1). 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.

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 alternative of 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. The rule for open(x) is similar.

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 [37]. 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. The already_dispatched(c) rule is expressed using the abbreviation [p,q) = !q S p. 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 AST of the formula, as it is represented internally by DEJAVU. The reader may compare it to the AST in Figure 2 for the first-order property (8). The AST (stored in now and pre) contains all the nodes needed for evaluation of the property. Two new kinds of nodes occur, compared to Figure 2, namely (purple) arrow shaped nodes (2 and 9) representing rule calls, and a (blue) node (5), with an additional small "file folder" extension on top of it, 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 that is called. This has the meaning that the "calling" (arrow-shaped) node will denote the same BDD value as the "called" node.

The property-specific part of the synthesized monitor[8] is shown in Figure 7. This function is called for each new event. The function operates on the two arrays holding BDDs, indexed by the subformula number: pre for the previous 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. It returns *true* (the property is satisfied in this position of the trace) iff now(0) is not BDD($\bot$), which effectively means that it is BDD($\top$) since the top-level formula contains no free variables. The evaluation uses the mixed evaluation or-

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

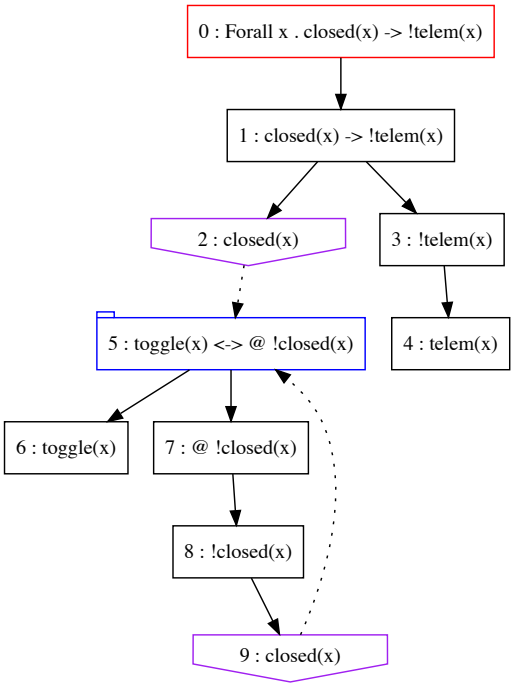


Fig. 6: Abstract Syntax Tree (AST) for the first-order property telemetry1 in Figure 5, illustrating the use of a rule. Two new kinds of nodes occur, compared to the AST in Figure 2, namely purple arrow shaped nodes (2 and 9) representing rule calls, and a blue node 5, with an additional small "file folder" extension on top of it, 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 that is called. This has the meaning that the "calling" (arrow-shaped) node will denote the same BDD value as the "called" node.

```
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.

der 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) -> !telem(x)) using the Boolean equivalence $(p \rightarrow q) \equiv (\neg p \vee q)$.

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 encountering the first event $toggle(L)$, the value L is assigned the 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. 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 negating the BDD from node 6, 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 see this, observe that node 7 is BDD($\bot$), since initially @!closed(x) (node 7) is BDD($\bot$) in the first state, and using the Boolean equivalence $(\mathcal{B} \leftrightarrow false) \equiv \neg\mathcal{B}$.

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.

Experiments. In [27, 28] we performed experiments with DEJAVU without the rule extension, comparing with the MONPOLY tool [7], which supports a logic close to DEJAVU's. In [26] we experimented with DEJAVU's garbage collection capability. In this section we present experiments with the rule extension for the four properties in Figure 5 on seven traces, named $\mathbf{T}_1$,...,$\mathbf{T}_7$, of various lengths and complexity. 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-id' 'trace length' : 'time in seconds' '(events per millisecond)' 'memory use'). 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.

The traces $\mathbf{T}_1$,...,$\mathbf{T}_6$ are created with log generation programs specifically for stress testing the properties telemetry1, telemetry2, and spawning. The last event of each of these traces violates the property. The trace $\mathbf{T}_7$ is a real log produced by the MSL rover [37]. The telemetry1 and telemetry2 properties (alternative formulations of the same property) are each verified against three traces $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$, of increasing lengths. These traces are produced by a trace generator $\mathbf{F}(r,c,t)$ with the following meaning. Repeat r times the following: toggle c channels to be open, send telemetry t times on each channel, toggle the c channels back to being closed. Finally add an event violating the property (sending telemetry on a closed channel). This yields $r*(c+(c*t)+c)+1$ events. The traces are generated with the following calls: $\mathbf{T}_1 = \mathbf{F}(100,1000,10)$ yielding 1,200,001 events, $\mathbf{T}_2 = \mathbf{F}(1000,100,50)$ yielding 5,200,001 events, and $\mathbf{T}_3 = \mathbf{F}(1000,100,100)$ yielding 10,200,001 events.

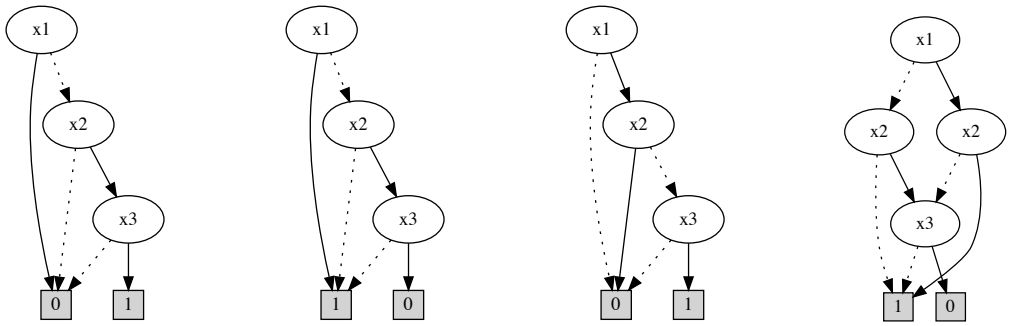
The spawning property is verified against the traces $\mathbf{T}_4$, $\mathbf{T}_5$, and $\mathbf{T}_6$ of increasing lengths. These traces are produced by a trace generator $\mathbf{G}(t,r)$ with the following meaning: spawn t threads from a main thread, let them each report back once to the main thread, repeat r times: each newly spawned thread in the previous iteration spawns a new thread, which then reports back to the main thread. Finally an event is inserted which violates the property (the main thread reports to itself). This yields $t+t+r*(t+t)+1$ events. The traces are generated with the following calls: $\mathbf{T}_4 = \mathbf{G}(49,100)$ yielding 9,899 events, $\mathbf{T}_5 = \mathbf{G}(99,100)$ yielding 19,999 events, and $\mathbf{T}_6 = \mathbf{G}(99,200)$ yielding 39,799 events.

Figure 9 illustrates events processed per millisecond for each combination of property and trace (short trace, medium trace, long trace), while Figure 10 illustrates maximal memory use in Megabytes. As can be seen, the spawning property performance, measured in processed events per ms, is worse than the properties telemetry1 and telemetry2. This is due in part to the need to store the transitive closure of the spawning operator, illustrated by the memory use in Figure 10. The commands property, applied to the MSL log, also performs worse than the telemetry properties. One observation is that the less well performing rules (spawning and commands) apply existential quantification in the rule bodies. Furthermore, the commands property has the most rules (three) of the properties. However, whether these are the actual reasons for the lower performance remains to be understood. One may conclude that the performance of the rule-based implementation is greatly dependent on property and trace.

DEJAVU is associated with an open source test suite consisting of 231 tests, focusing on functional correctness; 49 of these tests, involving 13 properties, target the rule-based extension. The traces for these functional correctness tests are usually small, focusing on function rather than performance. Schneider et. al. [41] performed randomized differential testing of DEJAVU as well as of their own MONPOLY tool. Using the Isabelle/HOL theorem prover, they developed a proven correct version of MONPOLY, named VERIMON. They generated 1000s of formulas automatically for MONPOLY as well as for DEJAVU, generated 1000s of traces influenced by the formulas, and then compared MONPOLY's and DEJAVU's results against those of VERIMON's, applied on those formulas and traces. In this exercise no algorithmic errors were detected in DEJAVU (a benign parsing error was detected for a corner case), whereas two algorithmic errors were detected in MONPOLY. Discrepancies, by design, were detected in DEJAVU's semantics and in VERIMON's semantics.

7 Conclusion

Propositional linear temporal logic (LTL) and automata are two common specification formalisms for software and



(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	Short traces	Medium traces	Long traces
telemetry1	$\mathbf{T}_1$ 1,200,001 : 2.6s (462/ms) 194 MB	$\mathbf{T}_2$ 5,200,001 : 5.9s (881/ms) 210 MB	$\mathbf{T}_3$ 10,200,001 : 10.7s (953/ms) 239 MB
telemetry2	$\mathbf{T}_1$ 1,200,001 : 3.8s (315/ms) 225 MB	$\mathbf{T}_2$ 5,200,001 : 8.7s (598/ms) 218 MB	$\mathbf{T}_3$ 10,200,001 : 16.6s (614/ms) 214 MB
spawning	$\mathbf{T}_4$ 9,899 : 29.5s (0.3/ms) 737 MB	$\mathbf{T}_5$ 19,999 : 117.3s (0.2/ms) 1,153 MB	$\mathbf{T}_6$ 39,799 : 512.5s (0.1/ms) 3,513 MB
commands	$\mathbf{T}_7$ 49,999 : 1.5s (33/ms) 169 MB	N/A	N/A

Table 1: Experiments - traces ($\mathbf{T}_1$,...,$\mathbf{T}_7$), trace lengths, analysis time in seconds, events per millisecond, and maximal memory use.

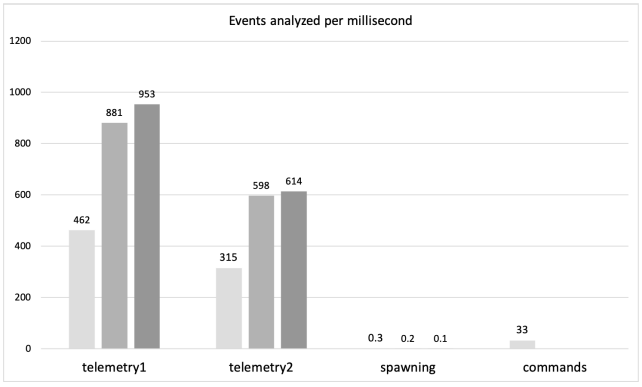


Fig. 9: Performance measured in events per millisecond for the four properties telemetry1, telemetry2, spawning, and commands on the seven traces $\mathbf{T}_1$,...,$\mathbf{T}_7$. The graph visualizes the events/ms numbers in Table 1. For each property (except the command property) is shown the performance on the three traces of increasing lengths that it was applied to (the darker the column the longer the trace). The command property was applied to one (realistic) trace.

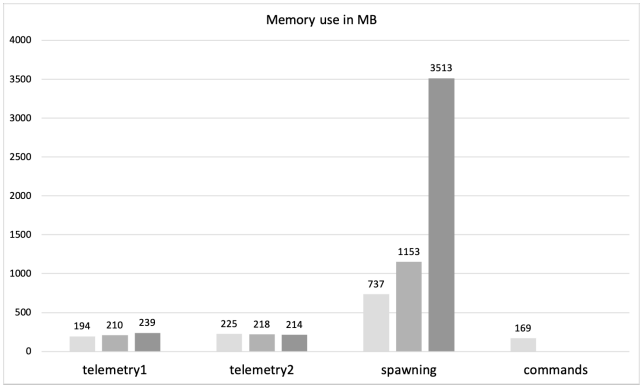


Fig. 10: Maximal memory use in MB for the four properties telemetry1, telemetry2, spawning, and commands on the seven traces $\mathbf{T}_1$,...,$\mathbf{T}_7$. The graph visualizes the MB numbers in Table 1. For each property (except the command property) is shown the performance on the three traces of increasing lengths that it was applied to (the darker the column the longer the trace). The command property was applied to one (realistic) trace.

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 based on the DEJAVU tool and provided experimental results. The code and many more examples appear at [16]. Future work includes making further comparisons between the different version of first order LTL logics and to other formalisms, in particularly, formalisms that are based on automata. 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, International Symposium on Temporal Representation and Reasoning (TIME'05), IEEE, 166-174. 2005.
3. D. P. Attard, I. Cassar, A. Francalanza, L. Aceto, A. Ingólfsdóttir, A runtime monitoring tool for actor-based systems, In Behavioural Types: from Theory to Tools (Simon Gay and Antonio Ravara eds.), Chapter 3, pp. 49-76, River Publishers, 2017.
4. H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS Volume 2937, Springer, 44-57, 2004.
5. H. Barringer, D. Rydeheard, K. Havelund, Rule systems for run-time monitoring: from Eagle to RuleR, Journal of Logic and Computation (JLC), Vol. 20 No. 3. Oxford University Press, 675–706, 2008.
6. 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.
7. 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.
8. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring metric first-order temporal properties, Journal of the ACM 62(2), 1-45, 2015.
9. A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly?, Workshop on Runtime Verification (RV'07), LNCS Volume 4839, Springer, 126-138, 2007.
10. J. Bohn, W. Damm, O. Grumberg, H. Hungar, K. Laster, First-order CTL model checking, International Conference on Foundations of Software Technology and Theoretical Computer Science, (FSTTCS'98), LNCS Volume 1530, Springer, 283-294, 1998.
11. R. E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, ACM Computing Survey 24(3), 293-318, 1992.
12. 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).
13. J. Chomicki, Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2): 149-186, 1995.
14. E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT Press, 2000.
15. N. Decker, M. Leucker, D. Thoma, Monitoring modulo theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 205-225, 2016.
16. DejaVu, https://github.com/havelund/dejavu.
17. H.-D. Ebbinghaus, J. Flum, W. Thomas, Mathematical Logic. Undergraduate texts in mathematics, Springer 1984.
18. Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime? STTT 14(3), 349-382, 2012.
19. C. Forgy, Rete: A Fast algorithm for the many pattern/-many object pattern match problem, Artificial Intelligence, Volume 19, 17-37, 1982.
20. H. Frenkel, O. Grumberg, S. Sheinvald, An automata-theoretic approach to modeling systems and specifications over infinite data, NASA Formal Methods (NFM'17), LNCS Volume 10227, Springer, 1-18, 2017.

21. J. Geist, K.Y. Rozier, J. Schumann, Runtime observer pairs and Bayesian network reasoners on-board FPGAs: Flight-certifiable system health management for embedded systems, International Conference on Runtime Verification (RV'14), LNCS Volume 8734, Springer, 215-230, 2014.
22. R. Gerth, D. A. Peled, M. Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic. PSTV 1995: 3-18.
23. S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
24. K. Havelund, Data automata in Scala, Theoretical Aspects of Software Engineering (TASE'14), IEEE Computer Society, 1-9, 2014.
25. K. Havelund, Rule-based runtime verification revisited, Software Tools for Technology Transfer 17(2), 143-170, 2015.
26. K. Havelund, D. Peled, Efficient runtime verification of first-order temporal properties. International Symposium on Model Checking Software (SPIN'18), LNCS Volume 10869, Springer, 26-47, 2018.
27. K. Havelund, D. A. Peled, D. Ulus, First-order temporal logic monitoring with BDDs, Formal Methods in Computer Aided Design (FMCAD'17), IEEE, 116-123, 2017.
28. K. Havelund, D. A. Peled, D. Ulus, First-order temporal logic monitoring with BDDs, Formal Methods in System Design: 1-21, 2019.
29. K. Havelund, G. Reger, D. Thoma, E. Zălinescu, Monitoring events that carry data, Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS Volume 10457, Springer, 61-102, 2018.
30. K. Havelund, G. Rosu, Synthesizing monitors for safety properties, Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), LNCS Volume 2280, Springer, 342-356, 2002.
31. L. Hella, L. Libkin, J. Nurmonen, L. Wong, Logics with aggregate operators. J. ACM 48(4): 880-907, 2001.
32. JavaBDD, http://javabdd.sourceforge.net.
33. O. Kupferman, M. Y. Vardi, Model checking of safety properties. Formal Methods in System Design 19(3): 291-314, 2001.
34. O. Maler, D. Ničković, Monitoring properties of analog and mixed-signal circuits. International Journal on Software Tools for Technology Transfer 15, Springer, 247–268, 2013.
35. Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, 1992.
36. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, International Journal on Software Tools for Technology Transfer 14, Springer, 249-289, 2012.
37. Mars Science Laboratory (MSL) mission website. http://mars.jpl.nasa.gov/msl.
38. L. Pike, N. Wegmann, S. Niller, A. Goodloe, Copilot: Monitoring embedded systems, Innovations in Systems and Software Engineering 9(4), 235-255, 2013.
39. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850TM-2010, 2010.
40. G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at runtime with QEA, Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), LNCS Volume 9035, Springer, 596-610, 2015.
41. J. Schneider, D. Basin, S. Krstić, D. Traytel, A formally verified monitor for metric first-order temporal logic, International Conference on Runtime Verification (RV'19), LNCS Volume 11757, Springer, 310–328, 2019.
42. N. Shafiei, K. Havelund, P. C. Mehlitz, Actor-based runtime verification with MESA, International Conference on Runtime Verification (RV'20), LNCS Volume 12399, Springer, 221-240, 2020.
43. A. P. Sistla, Theoretical Issues in the Design and Analysis of Distributed Systems, Ph.D Thesis, Harvard University, 1983.
44. W. Thomas, Automata on infinite objects, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, 133-192, 1990.
45. P. Wolper, Temporal logic can be more expressive, Information and Control 56(1/2): 72-99, 1983.
46. P. Wolper, M. Y. Vardi, A. P. Sistla: Reasoning about infinite computation paths, Annual Symposium on Foundations of Computer Science (SFCS'83), ACM, 185-194.