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

Klaus Havelund [⋆1] and Doron Peled [⋆⋆2]

[1] Jet Propulsion Laboratory, California Institute of Technology, USA
[2] Department of Computer Science, Bar Ilan University, Israel

Abstract. Runtime Verification (RV) consists of analyzing execution traces using formal techniques, e.g., monitoring executions against Linear Temporal Logic (LTL) properties. Propositional LTL is, however, limited in expressiveness, as first shown by Wolper [32]. Several extensions to propositional LTL, which promote the expressive power to that of regular expressions, have therefore been proposed; however, none of which was, by and large, adopted for RV. In addition, for many practical cases, there is a need in RV to monitor properties that carry data. This problem has been addressed by numerous authors, and in previous work we addressed this by providing an algorithm that uses BDDs to represent relations over data elements. We show expressiveness deficiencies of first-order LTL and suggest an extension of (propositional as well as first-order) LTL with rules to address these limitations. We describe how the DEJAVU tool is correspondingly extended and provide some experimental results.

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 in a formal logic, in our case 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 specification is usually performed by first translating the specification into a Büchi automaton. The automata formalisms are more expressive than LTL, with a classical example by Wolper [32], showing 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

⋆ The research performed by this 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 this author was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

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 therefore first present an alternative extension of propositional LTL with *rules*, named RLTL. These rules define and use auxiliary propositions, not appearing in the execution itself. These propositions obtain their values in a 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 to existing RV algorithms that use incremental summaries of prefixes, e.g., the classical algorithm [21] 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 equivalent to QLTL and its restriction to past properties is equivalent to Büchi automata and regular expressions.

Another expressiveness dimension is runtime verification of events that carry data, for which a first-order LTL supporting *quantification over data* is appropriate, here referred to as FLTL. We demonstrate the weakness of FLTL in expressing Wolper's example, relativized to the first-order case, and in expressing the transitive closure of temporal relations over events. We therefore 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 to the propositional case, the extension of the logic with quantification is more expressive than the extension with rules.

Runtime verification is commonly restricted to the *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 [21, 18], as opposed to future temporal logics [25]. The RV algorithm, presented here for RPFLTL (the safety part of RFLTL) naturally extends the RV algorithm for PFLTL in [18] in the same way that the algorithm we present for RPLTL (the safety part of RLTL) extends the RV algorithm in [21] for PLTL.

We further present a corresponding extension of the DEJAVU tool [18, 19, 17], 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 a Boolean enumeration of the input data (with a hash table representing the correspondence between the data and the enumerations). This is a very different use of BDDs from the classical model checking representation of sets of Boolean states[3]. A garbage collection algorithm tailored for that representation also assists in obtaining efficiency.

[3] E.g., in [6], BDDs are used to represent sets of program locations, and the data elements are represented symbolically as a formula.

Our main contribution is the LTL logics extended with rules (extensions prefixed with 'R'), and in particular the logic RPFLTL and its implementation. The structure of the 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.

2 Propositional LTL

The classical definition of 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. The models for LTL formulas are infinite sequence of states, of the form $\sigma = s_1 s_2 s_3 \ldots$, where $s_i \subseteq P$ for each $i \geq 1$. 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 $p \in s_i$.
- $(\sigma, i) \models \neg\varphi$ 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: $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\Diamond\varphi = (\mathit{true}\, \mathcal{U}\, \varphi)$, $\Box\varphi = \neg\Diamond\neg\varphi$, $\mathbf{P}\ \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ (**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 regular expressions over the alphabet $\Sigma = 2^P$ and to *monadic* first and second-order logic. Accordingly, we have the following characterizations: LTL is equivalent to monadic first-order logic, star-free regular expressions[4] and counter-free 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 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]. When PLTL is interpreted over finite sequences, its expressive power is the same as star-free regular expressions, first-order monadic logic over finite sequences and counting-free automata. Wolper [32] demonstrated that the expressive power of LTL is lacking using the property that all the states with even[5] indexes in a sequence satisfy some proposition p.

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 $\sigma|_X = s_1 \setminus X\, s_2 \setminus X \ldots$. (Note that $\sigma|_X$ 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 $\sigma'|_{\{q\}} = \sigma$ and $(\sigma', i) \models \varphi$.

Universal quantification is also allowed, where $\forall q\, \varphi = \neg \exists q \neg \varphi$. 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 regular expressions, full Büchi automata, or monadic second-order logic with unary predicates over the naturals (see again [31]). In fact, it is sufficient to restrict the quantification to existential quantifiers that prefix the formula 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 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 very natural for runtime verification.

[4] Regular expressions without the star operator (or ω).

[5] This is different than stating that p alternates between *true* and *false* on consecutive states.

We partition the propositions P into *auxiliary propositions* $A = \{a_1, \ldots, a_n\}$ and *basic propositions* B. An RLTL property η has the following form:

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

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 $\ominus$ operator. 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). The semantics can be defined as follows.

$$\sigma \models \eta \text{ if there exists } \sigma', \text{ where } \sigma'|_A = \sigma \text{ s.t. } \sigma' \models (\psi \wedge \Box \bigwedge_{1 \leq j \leq n} (a_j \leftrightarrow \varphi_j))$$

RLTL extends the set of propositions with new propositions, whose values at a state are functions of (i.e., uniquely defined by) the prefix of the model up to that state. This differs from the use of auxiliary propositions in QLTL, where the values assigned to the auxiliary propositions do not have to extend the states of the model in a unique way throughout the interpretation of the property over a model. 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:

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

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 of an execution by the prefix of the execution up to and including that state.*

Proof. Let η be a formula over auxiliary propositions A and basic propositions B, with rules $a_j := \varphi_j : j \in \{1, \ldots, n\}$. Let σ be a model with states over B. Then there is a unique model σ' such that $\sigma'|_A = \sigma$ and $\sigma' \models \Box \bigwedge_{1 \leq j \leq n} (a_j \leftrightarrow \varphi_j)$: inductively, 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'. $\Box$

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

Sketch of 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 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 by disallowing the future time temporal operators in RLTL. Every top level formula is interpreted as implicitly being prefixed with a $\Box$ operator, hence is checked in every state. This results in a formalism that is equivalent to a 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:*

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

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

In this form, the value of the Boolean variables a_j encodes 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 specifications often concentrates on the past portion of the logic. 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 this 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 to 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 order of calculating now for subformulas is bottom up, according to the syntax tree.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) := \mathit{false}$.
2. Observe a new event (as a set of propositions) s as input.
3. Let $\mathsf{pre} := \mathsf{now}$.
4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \mathit{true}$.
   - $\mathsf{now}(\varphi \wedge \psi) := \mathsf{now}(\varphi)$ *and* $\mathsf{now}(\psi)$.

- $\mathsf{now}(\neg\varphi) := \mathit{not}\ \mathsf{now}(\varphi)$.
- $\mathsf{now}(\varphi \mathcal{S} \psi) := \mathsf{now}(\psi)\ \mathit{or}\ (\mathsf{now}(\varphi)\ \mathit{and}\ \mathsf{pre}((\varphi \mathcal{S} \psi)))$.
- $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.

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 line 3 in the above algorithm). 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, as 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 $\mathsf{now}(\ominus\neg q) := \mathsf{pre}(\neg q)$.

Mixed evaluation order. Under mixed 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 of 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

Assume a finite set of infinite domains[6] $D_1, D_2, \ldots$, e.g., integers or strings. Let V be a finite set of *variables*, with typical instances x, y, z. An *assignment* over a set of variables V maps each variable $x \in V$ to a value from its associated domain $\mathit{domain}(x)$, where multiple variables (or all of them) can be related to the same domain. For example $[x \to 5, y \to$ "abc"$]$ assigns the values 5 to x and the value "abc" to y.

We define models for FLTL based on *temporal* relations [9], that is, relations with last parameter that is a natural number, representing a time instance in the execution. So a tuple of a relation R can be $(\text{“}a\text{”}, 5, \text{“}cbb\text{”}, 3)$, where 3 is the value of the time parameter. The last parameter i represents a discrete progress of time rather than modeling *physical real time*. It is used to allow the relations to have different tuples in different instances of i, corresponding to states in the propositional temporal logics.

For a relation R, $R[i]$ is the relation obtained from R by restricting it to the value i in the last parameter, and removing that last i from the tuples. For simplicity, we will describe henceforth the logic with relations R that have exactly two parameters, the second of which is the time instance. Hence $R[i]$ is a relation with just one parameter over a domain that will be denoted as $\mathit{dom}(R)$. The definition of the logic that allows relations with more parameters is quite straightforward. Our implementation, and the examples described later, fully support relations with zero or more parameters.

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

Syntax. The formulas of the core FLTL logic are defined by the following grammar, 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 model is a set of temporal relations $\mathcal{R} = \{R_1 \ldots, R_m\}$. Since the standard definition of temporal logic is over a sequence ("the execution"), let $\mathcal{R}[i] = \{R_1[i] \ldots, R_m[i]\}$. $\mathcal{R}[i]$ represents a *state*. A model $\mathcal{R}$ can thus be seen as a sequence of states $\mathcal{R}[1]\mathcal{R}[2]\ldots$. Let m be a bijection from relation names (syntax) to the relations $\mathcal{R}$ (semantics).

Let $\mathit{free}(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. We denote by $\gamma|_{\mathit{free}(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ. Let ε be the empty assignment (with no variables). In any of the following cases, $(\gamma, \mathcal{R}, i) \models \varphi$ is defined where γ is an assignment over $\mathit{free}(\varphi)$, and $i \geq 1$.

- $(\varepsilon, \mathcal{R}, i) \models \mathit{true}$.
- $(\varepsilon, \mathcal{R}, i) \models p(a)$ if $m(p)(a, i)$, where a denotes a constant from $\mathit{dom}(m(p))$.
- $([x \mapsto a], \mathcal{R}, i) \models p(x)$ if $m(p)(a, i)$, where $\mathit{domain}(x) = \mathit{dom}(m(p))$.
- $(\gamma, \mathcal{R}, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{\mathit{free}(\varphi)}, \mathcal{R}, i) \models \varphi$ and $(\gamma|_{\mathit{free}(\psi)}, \mathcal{R}, i) \models \psi$.
- $(\gamma, \mathcal{R}, i) \models \neg\varphi$ if not $(\gamma, \mathcal{R}, i) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models \bigcirc\varphi$ if $(\gamma, \mathcal{R}, i+1) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models (\varphi \ \mathcal{U} \ \psi)$ if for some j, $j \geq i$, $(\gamma|_{\mathit{free}(\psi)}, \mathcal{R}, j) \models \psi$ and for each k, $i \leq k < j$, $(\gamma|_{\mathit{free}(\varphi)}, \mathcal{R}, k) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \mathcal{R}, i-1) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models (\varphi \ \mathcal{S} \ \psi)$ if for some j, $1 \leq j \leq i$, $(\gamma|_{\mathit{free}(\psi)}, \mathcal{R}, j) \models \psi$ and for each k, $j < k \leq i$, $(\gamma|_{\mathit{free}(\varphi)}, \mathcal{R}, k) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models \exists x \, \varphi$ if there exists $a \in \mathit{domain}(x)$ such that[7] $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

For an FLTL (PFLTL) formula with no free variables, denote $\mathcal{R} \models \varphi$ when $(\varepsilon, \mathcal{R}, 1) \models \varphi$. 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 with $\exists$ and $\forall$, here 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. Let p and q be temporal 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 (executions) where only one data element a appears. Assume for the contradiction that there is an FLTL formula ψ that expresses this property. We recursively

[7] $\gamma[x \mapsto a]$ is the overriding of γ with the binding $[x \mapsto a]$.

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*; the *false* represents the Boolean value of $p(x)$ and $q(x)$ for any $x \neq a$, since only $p(a)$ and $q(a)$ may appear in the input. For example, $\exists x(q(x)\mathcal{S}p(x))$ becomes $(q_a\,\mathcal{S}\,p_a) \vee (\mathit{false}\,\mathcal{S}\,\mathit{false})$ (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. But Wolper's example [32] 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. Consider the property that asserts that when $\mathit{report}(y,x,d)$ appears in a state, denoting that process y sends some data d to a process x, there was a chain of process spawns: $\mathit{spawn}(x,x_1)$, $\mathit{spawn}(x_1,x_2)\ldots\mathit{spawn}(x_l,y)$. i.,e., y is a descendent process of x. The required property involves the transitive closure of the relation *spawn*. FLTL can be translated (in a way similar to the standard translation of LTL into monadic first-order logic formula [31]) to a first-order formula, with explicit occurrences of time variables over the naturals and the linear order relation $<$ (or $\leq$) between them. 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 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 relations, obtaining QFLTL (and the past-restricted version QPFLTL). The syntax includes $\exists p\,\varphi$, where p denotes a relation. We also allow $\forall p\ \varphi = \neg\exists p\neg\varphi$. The semantics is as follows.

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

Consequently, quantification over relations effectively extends the model $\mathcal{R}$ into a model $\mathcal{R}'$ within the scope of the quantifier. Note that quantification here is dynamic (as in QLTL and QPLTL) since the relations are temporal and can have different sets of tuples in different states.

Extending FLTL with rules. We now extend FLTL into RFLTL in a way that is motivated by the propositional extension from LTL (PLTL) to RLTL (RPLTL). We allow the following formula:

$$\psi \textit{ where } r_j(x_j) := \varphi_j(x_j) : j \in \{1,\ldots,n\} \quad \text{such that,} \tag{4}$$

1. ψ, the *statement*, is an FLTL formula with no free variables,
2. φ_j are PFLTL formulas with a single[8] free variable x_j,
3. r_j is an auxiliary temporal relation with two parameters: the first parameter is of the same type as x_j and the second one is, as usual, a natural number that is omitted

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

in the temporal formulas. An auxiliary relation r_j can appear within ψ. They can also appear in φ_k of a *rule* $r_k := \varphi_k$, but only within the scope of a previous-time operator $\ominus$.

We define the semantics for the RFLTL (RPFLTL) specification (4) by using the following equivalent QFLTL (QPFLTL, respectively) formula[9]:

$$\exists r_1 \ldots \exists r_n \, (\psi \wedge \Box \bigwedge_{j \in \{1,\ldots,n\}} (r_j(x_j) \leftrightarrow \varphi_i(x_j)) \tag{5}$$

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.*

Proof. By a simple induction, similar to Lemma 1. □

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

$$\Box \forall x \, (r(x) \rightarrow p(x)) \text{ where } r(x) = (q(x) \leftrightarrow \ominus \neg r(x)) \tag{6}$$

The property that corresponds to Example 2 appears as the property spawning in Figure 1 in the implementation section 6.

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

Sketch of Proof. 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, 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 there are no two consecutive data sets that are equivalent. This property can be expressed in QPFLTL.

We use a combinatorial argument to show by contradiction that one cannot express this property using any RPFLTL formula φ. The reason is that every prefix of a model for an RPFLTL property is extended uniquely with auxiliary relations, according to Lemma 4. Each prefix can be summarized by a finite number of relations: the ones in the model, the auxiliary relations and the assignments satisfying the subformulas. The size of each such relation is bounded by $O(m^N)$ where m is the number of values appearing in the prefix, and N is the number of parameters of the relations. However, the number of different data sets over m values is 2^m. This means that with large enough number of different values, each RPFLTL formula φ over the models of this property can have two prefixes with the same summary, where one of them has a data set that the other one does not. The semantics of RPFLTL implies that extending two prefixes with

[9] Formal semantics can also be given by constructing a set of temporal relations extended with the auxiliary ones inductively over growing prefixes.

the same summary in the same way would have the same truth value. Consequently, we can extend the two prefixes where some data set appears in one of them but not in the other into a complete model, and φ will not be able to distinguish between these models. □

From Theorem 2 and Equation (5) we immediately obtain:

Corollary 1. *Restricting the quantification of QPFLTL to existential quantification, strictly weakens its expressive power*[10].

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. (A typical use of runtime verification restricts the events for each state to a single event.) In our notation, the input consists of a sequence $\mathcal{R}[1]\,\mathcal{R}[2]\ldots$, which we earlier identified with states, where each $\mathcal{R}[i]$ consists of the relations in $\mathcal{R}$ with the last parameter is restricted to i. The RV algorithm will make use of sets of assignments over a set of variables, satisfying a subformula at some state (and stored in pre and now), also represented as relations (instead of propositions, as used for LTL in Section 3).

Set Semantics. The RV algorithm for (R)PLTL, presented in Section 3 calculates $\mathsf{now}(\varphi)$, for φ a subformula of the monitored property, to be the Boolean truth value of φ over the prefix inspected by the RV algorithm so far. For (R)PFLTL, $\mathsf{now}(\varphi)$ denotes the set of assignments satisfying φ (in the form of relations over the free variables in the subformula), rather than a Boolean value. We provide an alternative *set semantics* for the logic RPFLTL, without changing its interpretation, in a way that is more directly related to the calculation of values in now by the RV algorithm that will be presented below. Under the 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 $(\gamma,\sigma,i) \models \varphi$. We present here only two simple cases of the set semantics.

- $I[(\varphi \wedge \psi),\sigma,i] = I[\varphi,\mathcal{R},i] \bigcap I[\psi,\sigma,i]$.
- $I[(\varphi\ \mathcal{S}\ \psi),\mathcal{R},i] = I[\psi,\mathcal{R},i] \bigcup (I[\varphi,\mathcal{R},i] \bigcap I[(\varphi \mathcal{S} \psi),\mathcal{R},i-1])$.

Runtime verification algorithm for PFLTL. We start by describing an algorithm for monitoring PFLTL properties, presented in [18] and implemented in the tool DEJAVU. We enumerate data values appearing in monitored events, as soon as we first see them. We represent relations over the Boolean encoding of these enumeration, rather than over the data values themselves. A hash function is used to connect the data values to their enumerations to maintain consistency between these two representations. The relations are represented as BDDs [7]. For example, if the runtime-verifier sees the input events *open*("a"), *open*("b"), *open*("c"), it will encode the argument values as 000, 001 and 010 (say, we use 3 bits b_0, b_1 and b_2 to represent each enumeration, with b_2 being the

[10] It is interesting to note that for QPLTL, restriction to existential quantification does not change the expressive power.

most significant bit). A Boolean representation of the *set* of values {"a","b"} would be equivalent to a Boolean function $(\neg b_1 \wedge \neg b_2)$ that returns 1 for 000 and 001.

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 have just one enumeration representing these values per each variable of the LTL formula. We guarantee that at least one such enumeration exists by preserving 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, consult [17].

Given some ground predicate $p(a)$, observed in the monitored execution, matching with $p(x)$ in the monitored property, let $\mathbf{lookup}(x,a)$ be the enumeration of a (a lookup in the hash table). If this is a's first occurrence, then it will be assigned a new enumeration. Otherwise, **lookup** returns the enumeration that a received before. 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 Boolean representation. The function $\mathbf{build}(x,A)$ returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) v for $v \in A$. 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 we use the three Boolean variables (bits) x_0, x_1 and x_2 for representing enumerations over x (with x_0 being the least significant bit), and assume that $A = \{a,b\}$, $\mathbf{lookup}(x,a) = 000$, and $\mathbf{lookup}(x,b) = 001$. Then $\mathbf{build}(x,A)$ is a BDD representation of the Boolean function $(\neg x_1 \wedge \neg x_2)$.

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 $\mathbf{exists}(\langle x_0,\ldots,x_{k-1}\rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x\ \varphi$ in the current state. Finally, $\mathrm{BDD}(\bot)$ and $\mathrm{BDD}(\top)$ are the BDDs that return always 0 or 1, respectively. The algorithm for monitoring a formula η is as follows.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) := \mathrm{BDD}(\bot)$.
2. Observe a new state (as a set of ground predicates) s_i as input.
3. Let $\mathsf{pre} := \mathsf{now}$.
4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \mathrm{BDD}(\top)$.
   - $\mathsf{now}(p_k(a))$:= if $R_k[i](a)$ then $\mathrm{BDD}(\top)$ else $\mathrm{BDD}(\bot)$.
   - $\mathsf{now}(p_k(x)) := \mathbf{build}(x, \{a \mid R_k[i](a)\})$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) := \mathbf{not}(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) := \mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi\ \mathcal{S}\ \psi))))$.
   - $\mathsf{now}(\ominus\ \varphi) := \mathsf{pre}(\varphi)$.

– $\mathsf{now}(\exists x\, \varphi) := \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathsf{now}(\varphi))$.

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

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, as part of the runtime verification, as per the above PFLTL algorithm. 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 $\mathbf{rename}(r_j(x_j), y)$. 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 $\mathsf{now}(\varphi_j)$;
  $\mathsf{now}(r_j) := \mathsf{now}(\varphi_j)$;
  $\mathsf{now}(r_j(y)) := \mathbf{rename}(r_j(x_j), y)$;
  $\mathsf{now}(r_j(a)) :=$ 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 free 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 marriage of BDDs and Boolean enumeration is in particular efficient, since collections of adjacent Boolean enumerations tend to compact well.

6 Implementation

DEJAVU is implemented in SCALA. 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 1 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 variant of formula 6, illustrating the use of a rule 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

```
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. 1: Properties stated in DEJAVU's logic

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 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.

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 1 on a collection of traces. Table 1 shows the analysis time (excluding time to compile the generated monitor) and maximal memory usage in MB for different traces (format is 'trace length : time / memory'). 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.

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

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. There has been several proposed extensions to LTL that extend its expressive power to that of related automata formalisms. We proposed here a simple extension for propositional LTL that adds auxiliary propositions that summarize the prefix of the execution, based on rules written as past formulas. Conceptually, this extension puts the specification 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, and is in particular appealing for runtime verification of past (i.e., safety) temporal properties, which already are based on summarizing the value of subformulas over observed prefixes. We demonstrated that first-order linear temporal logic (FLTL), which can be used to assert 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 performing additional experiments, and making further comparisons to other formalisms. We intend to study further extensions, exploring the space between logic and programming.

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, FM-CAD 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.