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

No Author Given

No Institute Given

Abstract. Runtime Verification (RV) consists of analyzing execution traces using formal techniques. It includes monitoring the execution of a system against properties formulated in Linear Temporal Logic (LTL). LTL offers a succinct notation for writing useful specification properties. However, it is limited in expressiveness in the propositional case, and several theoretic extensions have therefore been proposed. Furthermore, for many practical cases, there is a need to monitor properties that carry data, where one can use formalisms like first-order LTL. We show that first-order LTL has similar expressiveness limitations as the propositional version. We suggest here two related extensions for increasing the expressive power: one for propositional LTL and one for first-order LTL. These extensions have a simple incremental operational semantics that is more suitable for RV than previously suggested extensions. We show that the propositional extension has the same expressiveness as the classical extensions, and demonstrate its adoption for runtime verification of first-order safety properties. Finally, we expand the BDD-based runtime verification tool ANONYMOOSE[1] to support our extension and perform some experiments.

1 Introduction

Runtime verification (RV) [3, 13] very generally 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 state of the observed system. Processing can take numerous forms. 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.

Linear Temporal Logic (LTL) is a common specification formalism for reactive and concurrent systems. It is often used in e.g. 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 [20], 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 [21] showing that it is not possible to express in LTL that every even state in the sequence satisfies p. This has motivated extending LTL in various ways. We are mainly motivated by runtime verification of executions *with data*, for which a first-order temporal logic is appropriate. There, too, we show that expressiveness can be extended.

[1] The actual tool name is not given due to anonymous reviewing.

We present first an extension of propositional LTL that is based on using additional (auxiliary) propositions, not appearing in the execution, which are dependent on the past of the current execution. These propositions obtain their value in a state as a function of the prefix of the execution up and including that state through a past temporal formula. This can be seen as adding summary states to the temporal specification, resulting in a specification that is in between logic and finite automata representation. This extension, which we believe is conceptually simpler and more intuitive than other suggested extensions, is shown to be equivalent to the other common specification formalism: Büchi automata, regular expressions and quantified LTL.

We then proceed to first-order LTL. We relativise Wolper's example to the first order case to motivate the need for extending first-order LTL. Another example is based on the unexpressiveness of the transitive closure of a relation in first order logic, where, e.g., using a relation representing neighbors in a graph is not enough for representing that there is a path between nodes. We present an extension where auxiliary relations, not appearing in the execution, are added. While this extension is capable of expressing the above properties, we show that for the first-order version, this has less expressive power than adding full quantification over relations. This in distinction from adding quantification to propositional LTL, where prefixing existential quantification provides the full expressive power of, say, Büchi automata.

The extensions we proposed to the logics are in particular natural for runtime verification of past (safety) temporal properties, since the values of propositional and first-order LTL are based on calculating and updating a summary of the prefix of the execution. Runtime verification is often restricted to a *past* version of temporal logic [14], mainly because for past properties we can decide when the monitored property is violated after monitoring a finite prefix of it, while for future properties we may never know for sure. Consistent with that, we provide runtime verification algorithms for the past-time versions of the extended propositional and first-order logics. We further present an extension of the ANONYMOOSE tool (publicly available on github, hidden here because of anonymous reviewing) that realizes the extended first-order past temporal logic verification. The ANONYMOOSE tool [24, 23] allows runtime verification on past time first-order temporal logic over infinite domains (e.g., the integers, strings). It achieves efficiency by using BDD representation of enumerations of data, rather than over the data values themselves, and by using a garbage collection algorithm.

The contributions of this paper are both theoretical and practical. On the theory side, we present and study extensions for propositional and first order linear-temporal logics and show the relations between them and existing versions. On the practical side, we present RV algorithms for the extended logic, and show how to cast them specifically in an efficient BDD-based implementation. We present some experiments performed with our implementation and report on the results.

The structure of this paper is as follows. In Section 2 we first define the syntax and semantics of propositional temporal logic. We then review some of its classical extensions and present our own extension. In Section 3 we present a runtime verification algorithm for the extended past LTL. In Section 4 we review first order linear temporal logic, present our extension and study the relationships between the presented versions of first-order LTL. In Section 5 we present a runtime verification algorithm for the past

version of the extended first-order LTL. Section 6 describes the implementation and provides experimental results. Finally, Section 7 concludes the paper.

Related work. For propositional LTL, the logic ETL [21, 22] extends the set of temporal operators by using automata (or, equivalently a right linear grammar). Each automaton defines a temporal operator, where the input letters of that automaton correspond to future temporal subformulas. Each accepted run of such an automaton defines then the places where the subformulas need to hold. Another expressive-equivalent extension, in [21], adds dynamic, i.e., state-dependent, quantification over propositions to LTL.

Addition of relations to temporal databases is done for aggregation [15], where relations are calculated, based on the current state, and are used for further calculating functions (sums, maxima) over the sequence of states. Aggregations were also used in the runtime verification tool MonPoly [4]. Numerous other systems have been produced for monitoring execution traces eith data against formal specifications. These include e.g. [2, 9, 12, 18, 19].

Conventions. In this paper, we treat and present several versions of the logic LTL. To simplify the presentation, we prefix LTL with letters as follows:

- P restricts to P*ast* temporal operators.
- F allows F*irst-order* quantification over data assigned to variables that is uniform for all states. This quantification is referred to as *static* quantification.
- Q adds (second-order) Q*uantification* that is state-dependent. This is referred to as *dynamic* quantification.
- E allows E*xtending* the set of objects that occur in the input with new auxiliary predicates (for the propositional versions) or relations (for the first-order cases) using past temporal logic rules that define them.

2 Propositional LTL

The classical definition of linear temporal logic [17] 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:

- $(\sigma, i) \models \mathit{true}$.
- $(\sigma, i) \models p$ iff $p \in s_i$.
- $(\sigma, i) \models \neg\varphi$ iff $(\sigma, i) \not\models \varphi$.
- $(\sigma, i) \models (\varphi \wedge \psi)$ iff $(\sigma, i) \models \varphi$ and $(\sigma, i) \models \psi$.
- $(\sigma, i) \models \bigcirc\varphi$ iff $(\sigma, i+1) \models \varphi$.
- $(\sigma, i) \models (\varphi \,\mathcal{U}\, \psi)$ iff for some j, $j \geq i$, $(\sigma, j) \models \psi$, and for each k, $i \leq k < j$, $(\sigma, k) \models \varphi$.
- $(\sigma, i) \models \ominus\varphi$ iff $i > 1$ and $(\sigma, i-1) \models \varphi$.

– $(\sigma, i) \models (\varphi \mathcal{S} \psi)$ iff there exists j, $1 \leq j \leq i$, such that $(\sigma, j) \models \psi$ and for each k, $j < k \leq i$, $(\sigma, k) \models \varphi$.

Then $\sigma \models \varphi$ when $(\sigma, 1) \models \varphi$. We can use the following abbreviations: $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $(\varphi \leftrightarrow \psi) = ((\neg\varphi \wedge \neg\psi) \vee (\varphi \wedge \psi))$, $\Diamond\varphi = (\mathit{true}\ \mathcal{U}\ \varphi)$ and $\Box\varphi = \neg\Diamond\neg\varphi$, $\mathbf{P}\ \varphi = (\mathit{true}\ \mathcal{S}\ \varphi)$ (**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 logics where the temporal propositions correspond to unary predicates (over time). An overview appears in [20]. Accordingly, we have the following characterizations: LTL is equivalent to monadic first-order logic[2], star-free regular expressions[3] and counter-free Büchi automata[4]. In fact, restricting the temporal operators to the *future* operators $\mathcal{U}$ and $\bigcirc$ (and the ones derived from them $\Box$ and $\Diamond$) leaves the same expressive power and is commonly used.

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 $(\sigma, |\sigma|) \models \varphi$. Alternatively, it is also a common practice to use the past time restricted version, 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] over infinite sequences. (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 [21] showed that the expressive power of LTL is lacking with respect to some important properties. As a canonical example, he demonstrated that it is impossible to express that all the states with even indexes in a sequence satisfy p (this is different than saying that p alternates between *true* and *false* on consecutive states). Wolper's example property cannot be expressed in PLTL either. In order to extend the expressiveness of LTL, Wolper suggested to use linear grammars, or, alternatively, the ability to quantify over propositions, as shown below.

Extending LTL with dynamic quantification. Adding quantification over propositions, as done in [21], 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 variables in X.)

– $(\sigma, i) \models \exists q\, \varphi$ iff 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 variables can have different truth values depending on the states. It is also called *second-order* quantification, since

[2] First-order logic with unary predicates over the naturals, and the relation $<$.

[3] Regular expressions but without star operator (or ω).

[4] A counter-free language is a regular language for which 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 $xy^n z \in L$.

the quantification establishes the *set* of states in which a propositional variable 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 [20]. In fact, it is sufficient to restrict the quantification to existential quantifiers that prefix the formula to obtain the full expressiveness of QLTL [20].

Wolper's property can be expressed in QLTL as follows:

$$\exists q\,((\neg q \wedge \Box(q \leftrightarrow \bigcirc \neg q)) \wedge \Box(q \rightarrow p)) \tag{1}$$

Restricting QPLTL to the past modalities, one obtain the logic PQLTL. PQLTL has the same expressive power as regular expressions and finite automata. Equation (1) can be rewritten in PQLTL as:

$$\exists q\,\mathbf{H}((q \leftrightarrow \ominus \neg q) \wedge (q \rightarrow p)) \tag{2}$$

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 now another extension of LTL, which we call ELTL. As will be showed later, this extension is is very natural for runtime verification.

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

$$\psi \textit{ where } a_j := \varphi_j : j \in \{1,\ldots,n\} \tag{3}$$

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 B can only occur within the scope of a $\ominus$ operator. The semantics can be defined as follows. We refer to ψ as the *statement* of η and to $a_j := \varphi_j$ as a *rule*. In text, rules will be separated from each other by commas.

$\sigma \models \eta$ if there exists σ', where $\sigma'|_A = \sigma$ such that
$\sigma' \models (\psi \wedge \Box \bigwedge_{1 \le j \le n}(a_j \leftrightarrow \varphi_j))$

Wolper's example can be written in ELTL as follows:

$$\Box(q \rightarrow p) \text{ where } (q := \ominus \neg q) \tag{4}$$

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

ELTL extends the set of propositional variables with new variables, whose values at a state are functions of (i.e., uniquely defined by) the prefix of the model up to that state. The added propositions effectively summarize the past execution prefix up to and including the current state. This differs from the use of auxiliary variables in QLTL, where the values assigned to the auxiliary variables 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 variables appearing in the formulas φ_i must occur within the scope of a $\ominus$ operator is required to help preventing that the values of the auxiliary

variables are conflicting, as in $a_1 := \neg a_2$ and $a_2 := a_1$, which makes a formula that uses a_1 and a_2 uniformly *false*. Still, one can write unsatisfiable rules, e.g., by using the rule $a_1 := \mathit{false}$.

Lemma 1. *Let φ be an ELTL formula over the auxiliary propositions $A = \{a_1, \ldots, a_n\}$ and basic propositions B, with a set of rules $a_j := \varphi_j : j \in \{1, \ldots, n\}$. Let σ be a model with states over the B. Then there is at most one model σ' such that $\sigma'|_A = \sigma$ and $\sigma' \models \Box \bigwedge_{1 \leq j \leq n} (a_j \leftrightarrow \varphi_j)$*

Sketch of proof. By induction on the state index. Each auxiliary variable is defined, via a PLTL formula by the current state of the model over the values of the propositions B, and by the values of $A \cup B$ in previous states. □

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

Proof. Each ELTL formula η, as defined above, 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 translate the QLTL property first into a second-order monadic logic formula, then to a deterministic Muller automata and then construct an ELTL 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. The details of the constructions are given in the appendix. □

We define EPLTL by disallowing the future temporal operators (except an implied prefixing $\Box$) in ELTL. This results in a formalism that is equivalent to finite (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 EPLTL is the same as QPLTL.*

In fact, it is easy to show that EPLTL can be restricted to the following form, while retaining its expressive power.

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

with p being one of the auxiliary variables a_j. Yet, this may not be the preferred form of an EPLTL property, and the use of further past-time temporal operators, and a non trivial statement part may provide a more intuitive specification. The relative expressive power between the propositional temporal logics discussed in this section is summarized in Figure 1. An arrow from A to B means that logic B is more expressive than A. A double headed arrow shows same expressiveness

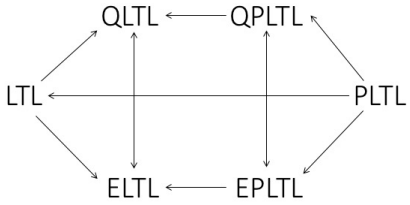

Fig. 1: Comparing the expressive power of the temporal logics in Section 2

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, 10].

The algorithm for PLTL first presented in [14] is based on the observation that the semantics of the past time formulas $\ominus\varphi$ and $(\varphi \mathcal{S} \psi)$ in the current step i is defined in terms of the semantics in the previous step $i-1$ of a subformula. To demonstrate this, we repeat here the definition of the previous-time operator $\ominus$, and rewrite the definition of $\mathcal{S}$ in an equivalent form that is more directly applicable for runtime verification.

- $(\sigma, i) \models \ominus\varphi$ iff $i > 1$ and $(\sigma, i-1) \models \varphi$.
- $(\sigma, i) \models (\varphi \mathcal{S} \psi)$ iff $(\sigma, i) \models \psi$ or the following hold: $i > 1$, $(\sigma, i) \models \varphi$ and $(\sigma, i-1) \models (\varphi \mathcal{S} \psi)$.

Runtime verification for a PLTL formula η exploits the fact that the semantic definition is recursive in both 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 (arrays) of values indexed by subformulas: pre, which summarizes the truth values of the subformulas for the execution prefix that ends just *before* the new 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 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* $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\neg\varphi) := \mathit{not}\ \mathsf{now}(\varphi)$.

- $\mathsf{now}(\varphi \mathcal{S} \psi) := \mathsf{now}(\psi)$ *or* $(\mathsf{now}(\varphi)$ *and* $\mathsf{pre}((\varphi \mathcal{S} \psi)))$.
- $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.

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

Runtime verification for QPLTL. This can be performed by translating a formula into a deterministic finite automaton, where one distinguished states, s, in a sink state. Then the sequence of states to be checked forms the input for this automaton, and violation is announced if s is reached. QPLTL can be extremely compact, in particular when using alternating existential and universal quantifiers, which can result in a huge (non-elementary) explosion of the automaton representing the same specification [20].

Runtime verification for EPLTL. For EPLTL, we need to add to the above algorithm calculations of $\mathsf{now}(a_j)$ and $\mathsf{now}(\varphi_j)$, for each of the rules 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 variables can appear recursively in EPLTL rules, the order of calculation is subtle. To see this, consider, for example, the formula (4). It contains the definition $q := \ominus \neg q$. Now, 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 $\mathsf{now}(\ominus \neg q) := \mathsf{pre}(\neg q)$.

Under *mixed evaluation order*, one calculates now as part of Step 4 of the above algorithm in the following order. The first four steps, *a-d*, are for the rules, and the fifth step *e* is for the statement η.

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. (This is not a problem, since $\mathsf{now}(\ominus \gamma)$ 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 $\mathsf{now}(a_j)$.
e. Calculate $\mathsf{now}(\eta)$ for the statement η.

4 First-Order LTL

Assume a finite set of infinite domains[5] $D_1, D_2, \ldots$, 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 $domain(x)$, 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.

[5] Finite domains are handled with some minor changes, see [24].

We define models for FLTL based on *temporal* relations [7]. That is relations with last parameter that is a natural number represented a time instance. So a tuple of a relation R can be $(\text{“}a\text{”}, 5, \text{“}cbb\text{”}, 3)$, where the value 3 is the time parameter. The last parameter i does not correspond to modeling physical time. This is rather 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, hence $R[i]$ will have just one parameter whose domain will be denoted as $d(R)$. The definition of the logic over other numbers of parameters is quite straightforward. e.g., when R has one parameter, i, $R[i]$ is a Boolean value. Our implementation of runtime verification described later fully supports relations with zero or more parameter. This also applies to the extensions of the logic in the rest of this section, and to some of the examples used.

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) onto 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 when 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 $d(m(p))$.
- $([x \mapsto a], \mathcal{R}, i) \models p(x)$ if $m(p)(a, i)$, where $\mathit{domain}(x) = d(m(p))$.
- $(\gamma, \mathcal{R}, i) \models (\varphi \wedge \psi)$ 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, $1 \leq j \leq i$, $(\gamma|_{free(\psi)}, \mathcal{R}, j) \models \psi$ and for each k, $j < k \leq i$, $(\gamma|_{free(\varphi)}, \mathcal{R}, k) \models \varphi$.
- $(\gamma, \mathcal{R}, i) \models \exists x\, \varphi$ if there exists $a \in domain(x)$ such that[6] $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

For an FLTL (PFLTL) formula with no free variables, denote $\mathcal{R} \models \varphi$ when $(\varepsilon, \mathcal{R}, 1) \models \varphi$. We will henceforth sometimes abuse notation, and use the same symbols both for the relations (semantics) and their representation in the logic (syntax). The quantification over values of variables here is *static* in the sense that they are independent of the state in the execution. We denote static quantification with $\exists$ and $\forall$.

We demonstrate that the lack of expressiveness carries over from LTL (PLTL) to FLTL (PFLTL).

Example 1. Let p and q be temporal relations, where their restrictions to the state s_i, $p[i]$ and $q[i]$, are unary relations p and q. 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). To show that this is not expressible in FLTL (and PFLTL), consider models (executions) where only one data element a appears. We can prove by a structural induction on subformulas that in this case, for each property φ we can replace occurrences of variables within predicates by a, i.e., $p(x)$ and $q(x)$ become $p(a)$ and $q(a)$, respectively, and that we can throw away the quantification ($\forall x$, $\exists x$), obtaining an equivalent formula. This is then equivalent to an LTL formula that is obtained by replacing the occurrences of $p(a)$ and $q(a)$ by propositions p_a and p_b, which reduces to Wolper's example [21]. Using parametric automata as a specification formalism, as in [11, 13, 18, 19], 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 $report(y,x,d)$ appears in a state, denoting that process y sends some data d to a process x, there was a chain of states with process spawns $spawn(x,x_1)$, $spawn(x_1,x_2)$ ... $spawn(x_l,y)$. i.,e., y is a descendent process of x. The reason is that the required property needs to assert about the transitive closure of the relation *spawn*. FLTL can be translated in a way similar to a standard translation of LTL to monadic first order logic [20]. Here, the relations will be written with their explicit last "time" parameter, and the temporal operators are replaced with first order quantification. E.g., $\Box \forall x\, p(x) \rightarrow q(y)$ will be translated into $\forall t\, \forall x\, ((p(x,t) \rightarrow \exists t' p(x,t'))$[7]. It is possible to show that the transitive closure of *spawn* cannot be expressed in first order setting. The proof is based on the compactness theory of first order logic [8].

Extending FLTL with dynamic quantification. Extending FLTL (PFLTL) with quantifiers over temporal relations, we obtain QFLTL (and the past-restricted version QPFLTL). The syntax includes $\exists p\, \varphi$, where p denotes an auxiliary relation. We also allow $\forall p\, \varphi = \neg \exists \neg p \varphi$. The semantics is as follows.

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

[7] The obtained model will have, in addition to the aforementioned relations, also the linear order $<$ over the naturals.

– $(\gamma, \mathcal{R}, i) \models \exists q\, \varphi$ iff 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 EFLTL in a way that is motivated by the propositional extension from LTL (PLTL) to ELTL (EPLTL). We allow the following formula:

$$\psi \textit{ where } r_j(x_j) := \varphi_j(x_j) : j \in \{1, \ldots, n\} \quad \text{such that,} \tag{5}$$

1. ψ, the *statement*, is an FLTL formula with no free variables,
2. φ_j are PFLTL formulas with a single free variable x_j,
3. r_j is an auxiliary temporal relation with 2 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 in the temporal formulas. An auxiliary relations 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[8] the semantics for the EFLTL (EPFLTL) specification (5) by using the following equivalent QFLTL (QPFLTL, respectively) formula:

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

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

Lemma 3. *The auxiliary temporal relations $r_1, \ldots r_n$ are uniquely defined by the rules for each model $\mathcal{R}$.*

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.

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

The property that corresponds to Example 2 appears as the third checked example in the implementation section 6.

Analogously to the propositional case, it is easy to show that EPFLTL does not need the past temporal operators besides $\ominus$.

Theorem 2. *The expressive power of EPFLTL is strictly weaker than that of QFLTL.*

[8] Formal semantics can also be given by constructing a set of temporal relations extended with the auxiliary ones inductively over growing prefixes, as would be done in a detailed proof of Lemma 3.

Sketch of Proof. The proof of this theorem includes encoding of a property that observes sets of data elements a, appearing separately, 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. The details appear in the appendix.

We use a combinatorial argument to show by contradiction that one cannot express this property in EPFLTL. The reason is that every prefix of an a model for an EPFLTL property is extended uniquely with auxiliary relations, according to Lemma 3. 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 EPFLTL formula over the models of this property can have two prefixes with the same summary where at least one of them has a data set that the other one does not. The semantics of EPFLTL implies that extending two models with the same summary in the same way would have the same truth value. So, we can extend the two models with a data set that appears in one of the prefixes but not in the other, and the EPFLTL property will not be able to distinguish between them. □

From Theorem 2 and Equation (6) we immediately obtain:

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

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. 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. A typical use of runtime verification restricts the events (tuples from the relations in $\mathcal{R}$) for each state to a finite number, and often even to a single event. The RV algorithm will also make use of sets of assignments over a set of variables for satisfying a subformula at some state, corresponding to pre and now, which had Boolean values in the algorithm for propositional LTL in Section 3.

Set Semantics. The RV algorithm for (E)PLTL, presented in Section 3 calculates $\mathsf{now}(\eta)$, for η a subformula of the monitored property, to be the truth value of φ over the prefix inspected by the RV algorithm so far. For (E)PFLTL, $\mathsf{now}(\eta)$ consists of the sets of assignments (in the form of relations over the free variables in the subformula), rather than Boolean variables.

We redefine the semantics for EPFLTL in equivalent way, which will be 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 [24] for PFLTL, and extended

here for EPFLTL), $I[\varphi,\sigma,i]$ is a function that returns 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. The full set semantics appears in the appendix.

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

Runtime verification algorithm for PFLTL. We start by describing an algorithm for monitoring PFLTL properties, first presented in [24] and implemented in the tool ANONYMOOSE. We enumerate data values appearing in monitored events, as soon as we first see them. We represent relations over the Boolean encodings 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 [6]. For example, if the runtime verifier sees the input events *open*("a"), *open*("b"), *open*("c"), it will encode them 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 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 (the value of b_0 can be arbitrary).

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 and their relations to all other values. 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 [23].

The use of BDDs can be replaced by other representations that can compactly and efficiently represent sets of values (e.g., ZDDs), and to which one can apply set operations like complementation, intersection, union (which are simply negation, conjunction and disjunction in BDDs) and projection (for the quantification). The marriage of BDDs and Boolean enumeration is in particular efficient, since collections of adjacent Boolean enumerations tend to compact well.

Given some ground predicate $p(a)$, observed in the monitored execution, matching with $p(x)$ in the monitored property, let **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 the value is converted to a Boolean representation.

The function **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 $A = \{a,b\}$,

lookup$(x,a) = 000$, and **lookup**$(x,b) = 001$. Then **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 to conjunction and disjunction of their BDD representation, respectively, and 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**$(\langle x_0,\ldots,x_{k-1}\rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x\ \varphi$ in the current state. Finally, BDD$(\bot)$ and BDD$(\top)$ are the BDDs that return always 0 or 1, respectively.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) := \text{BDD}(\bot)$.
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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \text{BDD}(\top)$.
   - $\mathsf{now}(p_k(a)) :=$ if $R_k[i](a)$ then BDD$(\top)$ else BDD$(\bot)$.
   - $\mathsf{now}(p_k(x)) :=$ **build**$(x, \{a \mid R_k[i](a)\})$.
   - $\mathsf{now}((\varphi \wedge \psi)) :=$ **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**$(\langle x_0,\ldots,x_{k-1}\rangle, \mathsf{now}(\varphi))$.
5. If $\mathsf{now}(\eta) = \mathit{false}$ then report a violation, otherwise goto step 2.

Runtime verification algorithm for EPFLTL We extend now the algorithm to capture EPFLTL. 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 FPLTL 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 $r_j(x_j) := \varphi_j(x_j)$ (this can be generalized to any number of variables), but r_j can be used as a subformula with other parameters in other rules or in the statement e.g., as $r_j(y)$. This can be resolved by a BDD renaming function **rename**$(r_j(x_j), y)$.

We then have to add following updates to step 4 of the above algorithm, when performing runtime verification of ψ.

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

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

6 Implementation

ANONYMOOSE is implemented in SCALA. ANONYMOOSE 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 [16].

Example properties. Figure 2 shows three properties in the input format of the tool, related to the examples in Section 4, which are not expressible in (P)FLTL. These properties are not expressible in the original first-order logic of ANONYMOOSE presented in [24]. The properties are The ASCII version of the logic here uses @ for $\ominus$, | for $\vee$, & for $\wedge$, and ! for $\neg$. The first property telemetry1 is a variant of formula 7, illustrating the use of a rule to express a first-order version of Wolper's example [21], that all the states with even indexes of a sequence satisfy a property. In this case we consider a radio on board a spacecraft, which communicates over different channels (quantified over in the formula), which 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, more closely 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 we in the previous state 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 spawning(x,y).

Evaluation. We evaluated the three properties in Figure 2 on a collection of traces. Table 1 shows the analysis time (excluding time to compile the generated monitor) for different traces (format is 'trace length : seconds'). The telemetry traces will repeatedly open hundreds of channels and send telemetry over them. The spawning traces consist of 50% spawning events and 50% reporting events. The spawning property requires larger processing time compared to the telemetry properties since more data (the transitive closure) has to be computed and stored.

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 declar-

```
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))
```

Fig. 2: Three properties stated in ANONYMOOSE's logic

Property	Trace 1	Trace 2	Trace 3
telemetry1	1,200,001 : 3.3s	5,200,001 : 7.1s	10,200,001 : 12.8s
telemetry2	1,200,001 : 4.4s	5,200,001 : 9.9s	10,200,001 : 17.9s
spawning	9,899 : 42.7s	19,999 : 127.6s	39,799 : 572.4s

Table 1: Evaluation - trace lengths and analysis time in seconds

ative 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 is based on adding propositions that summarize the prefix of an execution. 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 automata models, 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 extended it with relations that summarize prefixes. We showed 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, extending a classical algorithm, and similarly presented an algorithm for first-order past temporal logic. Finally we described the implementation of this extension in the ANONYMOOSE tool and provided experimental results. Future work includes performing additional experiments, and making further comparisons to other formalisms. Finally we would like 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. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Survety 24(3), 293-318 (1992).
7. J. Chomicki, Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. ACM Trans. Database Syst. 20(2): 149-186 (1995).
8. H.-D. Ebbinghaus, J. Flum, W. Thomas, Mathematical logic. Undergraduate texts in mathematics, Springer 1984.
9. 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.
10. Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime? STTT 14(3), 349-382, 2012.
11. H. Frenkel, O. Grumberg, S. Sheinvald, An Automata-Theoretic Approach to Modeling Systems and Specifications over Infinite Data. NFM 2017, 1-18.
12. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
13. 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.
14. K. Havelund, G. Rosu, Synthesizing monitors for safety properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
15. L. Hella, L. Libkin, J. Nurmonen, L. Wong, Logics with aggregate operators. J. ACM 48(4): 880-907 (2001).
16. JavaBDD, http://javabdd.sourceforge.net.
17. Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification. Springer, 1992.
18. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, STTT, Springer, 249-289, 2011.
19. 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.
20. W. Thomas, Automata on infinite objects, handbook of theoretical computer science, Volume B: Formal Models and Semantics, 133-192, 1990.
21. P. Wolper, Temporal logic can be more expressive, Information and Control 56(1/2): 72-99 (1983)

22. P. Wolper, M. Y. Vardi, A. P. Sistla: Reasoning about Infinite Computation Paths (Extended Abstract). FOCS 1983, 185-194.
23. Hidden due to anonymous submission.
24. Hidden due to anonymous submission.

A Appendix

Construction of a property for Theorem 2. The model consists of a temporal relation. r with only the time parameter, so $r[i]$ has a Boolean value at the ith state. We will henceforth say that r occurs in state i. Another relation is v, with two parameters. For each i, if r does not appear in the state, then there is a value a from some (infinite) domain, such that $v(a,i)$, i.e., $v(a)$ appears in i. The property requires that there are no two sets of values appearing within the relation v (as $v(a)$) between successive occurrences of r that are the same.

Hence, the sequence of states

$$\{r\}\{v(1)\}\{v(2)\}\{v(2)\}\{r\}\{v(2)\}\{v(1)\}\{v(5)\}\{r\}$$

is a model of this formula, while

$$\{r\}\{v(1)\}\{v(2)\}\{v(3)\}\{r\}\{v(2)\}\{v(1)\}\{v(3)\}\{r\}$$

is not. The QFLTL formula is a bit involved, hence given in several parts (macros), where the main formula appears last.

- $add(P) ::= \forall x\,(P(x) \leftrightarrow (v(x) \vee \ominus P(x))$ [The set P differs in the current state from the previous one only by values a that appear in $v(a)$.]
- $same(P) ::= \forall x\,(P(x) \leftrightarrow \ominus P(x))$ [The set of elements x for which $P(x)$ holds contains the same elements as in the previous state.]
- $empty(P) ::= \forall x \neg P(x)$ [The set P is empty.]
- $equal(P,Q) ::= \forall x(P(x) \leftrightarrow Q(x))$ [The sets represented by the relations P and Q contain the same elements *in the current state*.]
- $collect(P,\varphi,\psi) ::= r \wedge same(P) \wedge \varphi \wedge \ominus((add(P) \wedge \neg r \wedge \varphi)\ \mathcal{S}\ (r \wedge empty(P) \wedge \psi))$ [Collect elements a in $v(a)$, while φ holds, since the previous state where r and ψ held.]
- $init ::= r \wedge \Box(r \leftrightarrow \neg\exists x\, v(x))$ [r holds in the first state, and $v(a)$ with some value a holds in states where r does not hold.]
- $init \wedge \Box\neg\exists A \exists B(equal(A,B) \wedge$
  $collect(B, same(A), same(A))\ \mathcal{S}\ collect(A, true, true))$ [There are no two sequences of states where $v(a)$ holds with various values a, between a pair of states satisfying r that contain exactly the same elements (regardless of repetition).]

Details of left to right direction of Theorem 1.

One can translate any QLTL property into a second-order monadic logic in a standard way [20], and then from second-order monadic logic to a deterministic Muller automaton $M = (S, \iota, \Sigma, \Delta, \mathcal{F})$ (a Büchi automaton may not always be deterministic), where S is the set of states, $\iota \in S$ is the initial state, Σ is the set of inputs, each $\alpha \in \Sigma$ is a minterm (i.e., conjunction of each variable in the set negated or non-negated) of the variables in B, $\Delta : S \times \Sigma \mapsto S$ is the transition function, and $\mathcal{F} = \{F_1, \ldots, F_k\}$ is the accepting set. A muller automaton accepts a run if it passes infinitely often in exactly one of the specified accepting sets. To formulate an equivalent ELTL formula, the auxiliary variables represent the states S of M, i.e., a_s represents the state $s \in S$. Then we translate M into an ELTL formula of the form:

$\bigvee_{F\in\mathcal{F}} \bigwedge_{s\in F}(\Box\Diamond a_s \wedge \bigwedge_{s\notin F} \Diamond\Box\neg a_s)$ where
$a_s := (\alpha_{\mathfrak{l}} \wedge \ominus \mathit{false}) \vee \bigvee_{\Delta(s',\alpha)=s}(\alpha \wedge \ominus a_{s'}) : s \in S$

Basically, the assignments to the auxiliary variables are used to simulate the change of states[9], and the main formula expresses the Muller acceptance condition. □

Set Semantics. Let Γ be a set of assignments over the variables W, and $U \subseteq W$. Then $\mathit{hide}(\Gamma, U)$ (for "projecting *out*" or "hiding" the variables U) is the largest set of assignments over $W \setminus U$, each agreeing with some assignment of Γ on all the variables in $W \setminus U$. Let $U \cap W = \emptyset$, then $\mathit{ext}(\Gamma, U)$ is the largest set of assignments over $W \cup U$, where each such assignment agrees with some assignment in Γ on the values assigned to the variables W. Thus, if Γ is a set of assignments over W and Γ' is a set of assignments over W', then $\Gamma \bigcup \Gamma'$ is defined as $\mathit{ext}(\Gamma, W' \setminus W) \cup \mathit{ext}(\Gamma', W \setminus W')$ and $\Gamma \bigcap \Gamma'$ is $\mathit{ext}(\Gamma, W' \setminus W) \cap \mathit{ext}(\Gamma', W \setminus W')$. Hence, both are defined over the set of variables $W \cup W'$.

We denote by $A_{\mathit{free}(\varphi)}$ the set of all possible assignments of values to the variables that appear free in φ. Thus, $I[\varphi, \mathcal{R}, i] \subseteq A_{\mathit{free}(\varphi)}$. To simplify definitions, we add a dummy position 0 for sequence σ (which starts with s_1), where every formula is interpreted as an empty set. Observe that the value $\emptyset$ and $\{\varepsilon\}$, behave as the Boolean constants 0 and 1, respectively. The set semantics is defined as follows, where $i \geq 1$.

- $I[\varphi, \mathcal{R}, 0] = \emptyset$.
- $I[\mathit{true}, \mathcal{R}, i] = \{\varepsilon\}$.
- $I[p(a), \mathcal{R}, i] =$ if $m(p)(a,i)$ then $\{\varepsilon\}$ else $\emptyset$.
- $I[p(x), \mathcal{R}, i] = \{[x \mapsto a] | m(p)(a,i)\}$.
- $I[(\varphi \wedge \psi), \sigma, i] = I[\varphi, \mathcal{R}, i] \bigcap I[\psi, \sigma, i]$.
- $I[\neg\varphi, \mathcal{R}, i] = A_{\mathit{free}(\varphi)} \setminus I[\varphi, \mathcal{R}, i]$.
- $I[(\varphi \,\mathcal{S}\, \psi), \mathcal{R}, i] = I[\psi, \mathcal{R}, i] \bigcup (I[\varphi, \mathcal{R}, i] \bigcap I[(\varphi \mathcal{S} \psi), \mathcal{R}, i-1])$.
- $I[\ominus\varphi, \mathcal{R}, i] = I[\varphi, \mathcal{R}, i-1]$.
- $I[\exists x\, \varphi, \sigma, i] = \mathit{hide}(I[\varphi, \mathcal{R}, i], \{x\})$.
- $I[\psi \text{ where } r_j(x_j) := \varphi_j(x_j) : j \in \{1,\ldots,n\}, \mathcal{R}, i] = I[\psi, \mathcal{Q}]$ where $\mathcal{Q} = \mathcal{R} \cup \{r_j | r_j[i] = I[\varphi_j, \mathcal{Q}, i]\}$

Note that the last time interprets a formula of the form defined in (5). In the last item, the extended model $\mathcal{Q}$ appears both on the left and the right of the equality sign. It is well defined because of Lemma 3 (equivalently, a more verbose definition can include a stepwise construction of $\mathcal{Q}$ from $\mathcal{R}$).

[9] It is possible to use only $\lceil \log(|S|) \rceil$ variables, at the expense of longer and more complicated formulas.