From First-order Temporal Logic to Parametric Trace Slicing

Giles Reger and David Rydeheard

University of Manchester, Manchester, UK

Abstract. Parametric runtime verification is the process of verifying properties of execution traces of (data carrying) events produced by a running system. This paper considers the relationship between two widely-used specification approaches to parametric runtime verification: *trace slicing* and *first-order temporal logic*. This work is a first step in understanding this relationship. We introduce a technique of identifying *syntactic fragments* of temporal logics that admit notions of *sliceability*. We show how to translate formulas in such fragments into automata with a slicing-based semantics. In exploring this relationship, the paper aims to allow monitoring techniques to be shared between the two approaches and initiate a wider effort to unify specification languages for runtime verification.

1 Introduction

Runtime verification [12] is the process of checking properties of execution traces produced by running a computational system. An execution trace is a finite sequence of *events* generated by the computation. In many applications, events carry *data values* – the so-called parametric, or first-order, case of runtime verification. To formalise runtime verification, we need to provide (a) a specification language for describing properties of execution traces, and (b) a mechanism for checking these formally-defined properties during execution, i.e. a mechanism for generating monitors from specifications. Many different formalisms have been proposed (see Sec. 7). In fact, almost every runtime verification approach introduces its own specification language. One aim of this work is to develop techniques for relating approaches to runtime verification as a first step to bringing some order to this variety of formalisms.

One approach to runtime verification [3, 15] is to use automata both to specify trace properties and to act as monitors of execution traces. In the first-order case, the semantics of automata can be defined in terms of so-called *trace slicing* [9], whereby traces are projected according to the values carried by events, and properties are evaluated on these projections. This has been shown to be highly efficient [1]. An alternative approach is to use temporal logic to specify properties of traces. Mechanisms for constructing monitors from first-order temporal logic specifications have been proposed (see, for example, [20, 5, 7, 10, 14]). This paper considers the relationship between these two approaches.

In general, properties expressed in a first-order temporal logic do not respect trace slicing. We examine the reasons for this and then introduce a technique for exploring the relationship between trace slicing and temporal logic. To do so, we introduce a first-order linear temporal logic with a finite trace semantics. The key step is then to identify *syntactic fragments* of this logic and the corresponding notions of 'sliceability'. We give

one example of such a syntactic fragment and prove that it admits a notion of sliceability. We then show how we may construct monitoring automata from formulas in the fragment. To what extent do such syntactic fragments provide specification languages for runtime verification? As we discuss, currently the expressivity of formalisms *for runtime verification* is not easy to assess as we do not have adequate data on specifications likely to occur in runtime verification activities.

There are two main motivations behind this work. Firstly, giving a translation from first-order temporal logic into slicing-based formalisms allows specifications written in the former language to be monitored using techniques based on the latter. Secondly, the wide range of specification languages for (parametric) runtime verification often makes comparison and re-usability of specifications difficult. This work therefore is a contribution to unifying specification languages for runtime verification. More precisely, it is a *necessary first step* in the authors' wider goal of finding a correspondence between the full expressiveness of the slicing-based formalism QEA and temporal logics.

The contributions of this paper are as follows:

- Based on a first-order linear temporal logic (Sec. 2) and slicing (Sec. 3), we describe restrictions that slicing places on the structure of formulas, and introduce a syntactic fragment that satisfies these restrictions (Sec. 4)
- To make the correspondence practically useful we provide a translation from formulas in the fragment to a slicing-based formalism (Sec. 5)

The paper finishes with related work (Sec. 7) and conclusions (Sec. 8).

2 A First-Order Linear Temporal Logic

We begin by presenting a first-order discrete linear-time temporal logic, $\mathsf{FO\text{-}LTL}_f$, with a finite-trace semantics, where traces are finite sequences of events (see [8] for a discussion). As we are focussing on the correspondence with slicing, we do not consider general first-order functions and predicates; we plan to consider these in future work.

Let Σ be a finite set of names of events, Var be a finite set of variable names and Val be a finite set of value symbols (constants) disjoint from Var. An *event* $e(z_1, \dots, z_n)$ is an element of the set $\Sigma \times (Var \cup Val)^*$. An event is *ground* if all of its parameters are values. We write events as $\mathbf{a}, \mathbf{b} \dots$ A (ground) *trace* is a finite sequence of (ground) events. We write the empty trace as ϵ. Given a trace τ we write the length of a trace as $|\tau|$ and the i-th element as τ_i where the first element is at index 0.

The syntax of $\mathsf{FO\text{-}LTL}_f$ is defined as the following formulas:

$$\phi = \mathit{true} \mid \mathbf{a} \mid \forall x : \phi \mid \neg\phi \mid \phi \vee \phi \mid \phi\, \mathcal{U}^\circ\, \phi$$

We use standard identities, defining $\mathit{false} = \neg\mathit{true}$, $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$, $\exists x : \phi = \neg\forall x : \neg\phi$ and $\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$. The logic incorporates a single temporal modality $\mathcal{U}^\circ$ which can be read as *next until*. This is sufficient for defining in $\mathsf{FO\text{-}LTL}_f$ the temporal modalities that we would expect to see in a discrete linear-time temporal logic: the 'next' modality $\bigcirc\phi = \mathit{false}\, \mathcal{U}^\circ\, \phi$; 'until' $\phi_1\, \mathcal{U}\, \phi_2 = \phi_2 \vee (\phi_1 \wedge (\phi_1\, \mathcal{U}^\circ\, \phi_2))$; 'eventually' $\Diamond\phi = \mathit{true}\, \mathcal{U}\, \phi$; and 'always' $\Box\phi = \phi\, \mathcal{U}\, \mathit{false}$.

A *valuation* is a map (i.e. a partial function with finite domain) from variables to values. For valuations θ_1 and θ_2 let $\theta_1 \dagger \theta_2$ be the valuation where θ_2 *overrides* or

extends values for variables in θ_1. Valuations can be applied to events and to formulas to replace variables with values. A *domain* is a map from variables to sets of values. Let $\texttt{events}(\phi)$ be the set of events occurring in ϕ.

A formula ϕ is a *sentence* if it has no free variables. We define the semantics of $\mathsf{FO\text{-}LTL}_f$ in terms of a models relation $\models$ on sentences:

Definition 1 (Semantics). *We define the semantics of* $\mathsf{FO\text{-}LTL}_f$ *with respect to a quadruple* $(\mathcal{D}, \tau, v, i)$ *where* $\mathcal{D}$ *is the domains of variables,* τ *is a trace,* v *is a valuation and* i *an index of the trace. The relation* $\models$ *is defined as follows:*

$$
\begin{array}{ll}
\mathcal{D}, \tau, v, i \models \mathit{true} & \\
\mathcal{D}, \tau, v, i \models \mathbf{a} & \text{if } \tau_i = v(\mathbf{a}) \\
\mathcal{D}, \tau, v, i \models \neg\phi & \text{if } \mathcal{D}, \tau, v, i \not\models \phi \\
\mathcal{D}, \tau, v, i \models \phi_1 \vee \phi_2 & \text{if } \mathcal{D}, \tau, v, i \models \phi_1 \text{ or } \mathcal{D}, \tau, v, i \models \phi_2 \\
\mathcal{D}, \tau, v, i \models \phi_1\, \mathcal{U}^\circ\, \phi_2 & \text{if there exists a } j > i \text{ such that either } \mathcal{D}, \tau, v, j \models \phi_2 \text{ or} \\
 & \qquad j = |\tau| \text{ and } \phi_2 = \mathit{false} \\
 & \text{and for every } k \text{ where } i < k < j \text{ we have } \mathcal{D}, \tau, v, k \models \phi_1 \\
\mathcal{D}, \tau, v, i \models \forall x : \phi & \text{if for every } d \in \mathcal{D}(x) \text{ we have } \mathcal{D}, \tau, v \dagger [x \mapsto d], i \models \phi
\end{array}
$$

Linear temporal logics are usually defined on *infinite* traces. However, in runtime verification, we evaluate formulas on *finite* traces. We therefore consider how temporal properties should behave at the end of a trace. A common approach (see [18]) is to assume that *next* and *eventually* evaluate to false beyond the end of a finite trace and *always* evaluates to true. This captures the intuition that these modalities represent obligations for something desired to happen in the unfinished trace whereas the *always* modality captures an obligation for something undesired not to happen. We capture this idea with a special treatment of $\phi_1\, \mathcal{U}^\circ\, \phi_2$, where ϕ_2 is false. In this case, we allow the obligation to hold after the end of the trace. This gives the above trace semantics for the temporal modality $\mathcal{U}^\circ$.

We write $\tau \models \phi$ if a trace τ satisfies a property ϕ, defined as follows

$$\tau \models \phi \quad \text{iff} \quad \texttt{dom}(\tau, \phi), \tau, [], 0 \models \phi$$

where the domain function $\texttt{dom}$ is defined as:

$$\texttt{dom}(\tau, \phi)(x) = \{d_i \mid e(\ldots, d_i, \ldots) \in \tau \wedge e(\ldots, x_i, \ldots) \in \texttt{events}(\phi) \wedge x_i = x\}$$

Prefix semantics. An alternative way of viewing finite traces is as *prefixes* of infinite traces, leading to a multi-valued semantics based on whether the trace could be extended to an accepting infinite trace [8]. We do not consider this view here but note that QEA [3] has this notion of multi-valued verdicts based on possible extensions. We will explore this correspondence further in future work.

3 Parametric Trace Slicing

Parametric trace slicing [9] is a technique that transforms a monitoring problem involving quantification *over finite domains* into a propositional one. The idea is to take each valuation of the quantified variables and consider the specification *grounded* with that valuation for the trace *projected* with respect to the valuation. Ground events can then be considered as propositional symbols in the specification and in the projected traces. The benefit of this approach is that projection can lead to efficient indexing techniques.

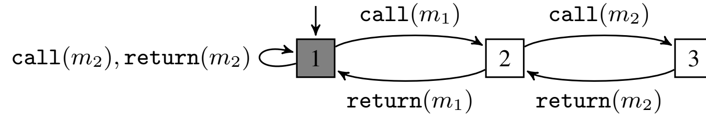
Introductory example. We illustrate the notion of trace slicing using an example of calls and returns of (non-recursive) methods. A required property is that whenever a method m_2 is called inside a method m_1, the method m_2 should return before m_1. This gives rise to a set of abstract events: $\texttt{call}(m_1), \texttt{return}(m_1), \texttt{call}(m_2), \texttt{return}(m_2)$. The property should hold *for all* values for m_1 and m_2 and is therefore a *quantified* property. To understand how trace slicing works consider the following trace:

$$\texttt{call}(A).\texttt{call}(B).\texttt{call}(C).\texttt{return}(C).\texttt{return}(B).\texttt{call}(C).\texttt{return}(C).\texttt{return}(A)$$

There are three values that m_1 and m_2 can take, A, B or C, and the trace is *sliced* with respect to each valuation of m_1 and m_2. The following table gives the trace slices for the non-equal values for m_1 and m_2, omitting symmetric cases.

m_1	m_2	slice
A	B	$\texttt{call}(A).\texttt{call}(B).\texttt{return}(B).\texttt{return}(A)$
A	C	$\texttt{call}(A).\texttt{call}(C).\texttt{return}(C).\texttt{call}(C).\texttt{return}(C).\texttt{return}(A)$
B	C	$\texttt{call}(B).\texttt{call}(C).\texttt{return}(C).\texttt{return}(B).\texttt{call}(C).\texttt{return}(C)$

Each slice can be checked against a quantifier-free property for a given m_1 and m_2. The above property is captured in the automaton below which processes the trace by replacing m_1 and m_2 appropriately for each slice.



3.1 Defining slicing

We define a variant of slicing as in [17], which is based on [9]. Let $\mathcal{A}(X)$ be an *event alphabet* (i.e. a set of events) where events use exactly those variables in the set X. A **quantifier-free property** $\mathcal{P}(X)$ over alphabet $\mathcal{A}(X)$ defines a language $\mathcal{L}(\mathcal{P}(X))$ over $\mathcal{A}(X)$. This can be grounded by giving values for X. Given a valuation θ with domain X let the grounded language $\mathcal{L}(\theta, \mathcal{P}(X))$ be given by θ applied to each event in each trace of $\mathcal{L}(\mathcal{P}(X))$. For example we may have $\texttt{a}(x).\texttt{b}(y) \in \mathcal{L}(\mathcal{P}(\{x, y\}))$ and therefore $\texttt{a}(1).\texttt{b}(2) \in \mathcal{L}([x \mapsto 1, y \mapsto 2], \mathcal{P}(\{x, y\}))$. Slicing can then be defined in terms of the ground events from the trace that match events in the event alphabet:

Definition 2 (Slicing). *Given a trace τ and valuation θ let $\tau \downarrow_\theta$ be the θ-slice of τ*

$$\begin{aligned} \epsilon \downarrow_\theta &= \epsilon \\ \tau.e(\overline{v}) \downarrow_\theta &= \begin{cases} (\tau \downarrow_\theta).e(\overline{v}) & \textit{if } \exists e(\overline{z}) \in \mathcal{A}(X) : \theta(e(\overline{z})) = e(\overline{v}) \\ (\tau \downarrow_\theta) & \textit{otherwise} \end{cases} \end{aligned}$$

A **quantified property** $\langle \Lambda(X), \mathcal{P}(X) \rangle$ consists of a list of quantifications (quantifiers and variables) and a quantifier-free property over a shared set of variables X. Some systems [15] only consider universal quantification and in this case acceptance is defined as the acceptance by $\mathcal{P}(X)$ of **all** θ-trace slices for all possible valuations θ. However, it is straightforward to introduce existential quantification also [3]. In this case the notion of acceptance captures the boolean combination of the quantifier-free acceptance for possible valuations.

Definition 3 (Acceptance). *The trace τ is accepted for quantification list $\Lambda(X)$ and propositional property $\mathcal{P}(X)$ if $\tau \models_{[]}^{\mathcal{P}(X)} \Lambda(X)$, defined as*

$$\begin{array}{lll} \tau \models_{\theta}^{\mathcal{P}(X)} \forall x : \Lambda & \textit{if for every } d \in \mathtt{dom}(x) \textit{ we have } \tau \models_{\theta\dagger[x\mapsto d]}^{\mathcal{P}(X)} \Lambda \\ \tau \models_{\theta}^{\mathcal{P}(X)} \exists x : \Lambda & \textit{if for some } d \in \mathtt{dom}(x) \textit{ we have } \tau \models_{\theta\dagger[x\mapsto d]}^{\mathcal{P}(X)} \Lambda \\ \tau \models_{\theta}^{\mathcal{P}(X)} \epsilon & \textit{if } \tau \downarrow_{\theta} \in \mathcal{L}(\theta, \mathcal{P}(X)) \end{array}$$

3.2 Choices for the quantifier-free language

The JAVAMOP system [15] is based on parametric trace slicing and introduces multiple languages for the quantifier-free part, including finite state automata and linear temporal logic. Quantified Event Automata (QEA) [3, 16] use a form of extended finite state machine that allows unquantified variables to capture changing values in the trace. Later (Sec. 8) we discuss how this can be used to extend the fragment of $\mathsf{FO\text{-}LTL}_f$ defined next. Note that QEA has a very efficient monitoring tool MARQ [17]. Here we use a simplified form of QEA that uses finite state automata as the quantifier-free formalism.

4 Temporal logic and trace slicing

In this section, we introduce a syntactic fragment of $\mathsf{FO\text{-}LTL}_f$ (see Sec. 2), and show how it relates to trace slicing. We begin by introducing the notion of *slicing invariance* and then discuss restrictions on $\mathsf{FO\text{-}LTL}_f$ formulas that respect this invariance. Finally, we describe a syntactic fragment that satisfies these restrictions.

4.1 Sliceability

A formula in $\mathsf{FO\text{-}LTL}_f$ is *sliceable* if its truth value for a valuation of its free variables is the same over a trace and the corresponding trace slice:

Definition 4. *A formula ψ with free variables X is* sliceable *if for valuation θ over X and trace τ*

$$\tau, \theta \models \psi \quad \Leftrightarrow \quad \tau \downarrow_{\theta}, \theta \models \psi.$$

We can phrase sliceability in terms of invariance with respect to *non-relevant events* i.e. the evaluation is stable under the deletion and addition of those events that are removed during slicing. The events relevant to a formula ψ, for valuation θ, are defined as $\mathsf{relevant}(\psi, \theta) = \{\theta(\mathbf{a}) \mid \mathbf{a} \in \mathsf{events}(\psi)\}$. Thus, a trace slice includes exactly those events relevant to the valuation. A formula is *slicing invariant* if adding/removing non-relevant events to/from a trace has no effect on whether the trace satisfies the formula:

Definition 5 (Slicing invariance). *Given a formula ψ with free variables X and valuation θ over X, let $\mathcal{L}(\psi, \theta) = \{\tau \mid \tau, \theta \models \psi\}$ be the set of traces that satisfy ψ. Let the* non-relevance-closure *of $\mathcal{L}(\psi, \theta)$ be inductively defined as the smallest set satisfying*

$$\begin{array}{lll} \tau & \in \mathcal{L}^C(\psi, \theta) & \textit{if } \tau \in \mathcal{L}(\psi, \theta) \\ \tau_1.\tau_2.\tau_3 & \in \mathcal{L}^C(\psi, \theta) & \textit{if } \forall \mathbf{a} \in \tau_2 : \mathbf{a} \notin \mathsf{relevant}(\psi, \theta) \textit{ and } \tau_1.\tau_3 \in \mathcal{L}^C(\psi, \theta) \\ \tau_1.\tau_3 & \in \mathcal{L}^C(\psi, \theta) & \textit{if } \exists \tau_1.\tau_2.\tau_3 \in \mathcal{L}^C(\psi, \theta) : \forall \mathbf{a} \in \tau_2 : \mathbf{a} \notin \mathsf{relevant}(\psi, \theta) \end{array}$$

The formula ψ is slicing invariant *if $\mathcal{L}(\psi, \theta) = \mathcal{L}^C(\psi, \theta)$.*

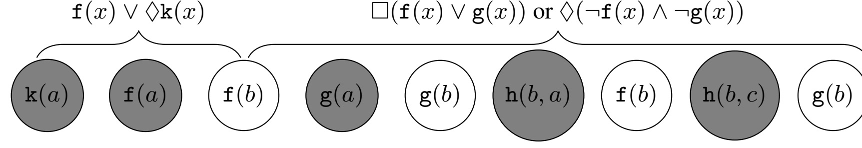


Fig. 1. Illustrating sliceable restrictions

Note that if we treat ground events as propositional symbols, as is common in slicing approaches, this invariance corresponds to removing/adding symbols from/to the trace that do not occur in the formula ψ.
We show that the notions of being sliceable and being slicing invariant are the same.

Lemma 1. *The sliceable and slicing invariant formulas coincide. For slicing invariant formula ψ over X, valuation θ over X and trace τ,*

$$\tau \in \mathcal{L}(\psi, \theta) \Leftrightarrow \tau \downarrow_\theta \in \mathcal{L}(\psi, \theta)$$

Proof. Firstly we note that as ψ is slicing invariant $\mathcal{L}(\psi, \theta) = \mathcal{L}^C(\psi, \theta)$ as per Def. 5. The proof proceeds by induction on the length of τ. For the base case where $\tau = \epsilon$ we have $\tau = \tau \downarrow_\theta$ and the property holds trivially. In general, note that if τ and $\tau \downarrow_\theta$ have the same length then τ contains only relevant events and $\tau = \tau \downarrow_\theta$. Therefore, we assume $\tau = \tau_1.\mathbf{a}.\tau_2$ for some non-relevant event $\mathbf{a}$

In the $\Rightarrow$ direction assume $\tau \in \mathcal{L}(\psi, \theta)$. As $\mathbf{a}$ is non-relevant and by the second rule in Def. 5 we have $\tau_1.\tau_3 \in \mathcal{L}(\psi, \theta)$. As $\tau_1.\tau_3$ is shorter than τ we can apply the induction hypothesis to conclude $(\tau_1.\tau_3) \downarrow_\theta \in \mathcal{L}(\psi, \theta)$. By definition $\tau \downarrow_\theta = (\tau_1 \downarrow_\theta)$. $(\tau_3 \downarrow_\theta) = (\tau_1.\tau_3 \downarrow_\theta)$ and the property holds. In the $\Leftarrow$ direction assume $\tau \downarrow_\theta \in \mathcal{L}(\psi, \theta)$. By definition $\tau \downarrow_\theta = \tau_1 \downarrow_\theta .\tau_3 \downarrow_\theta$ as $\mathbf{a}$ is non-relevant. As $\tau_1 \downarrow_\theta .\tau_3 \downarrow_\theta$ is shorter than τ we can apply the induction hypothesis to conclude that $\tau_1.\tau_3 \in \mathcal{L}(\psi, \theta)$. Therefore, by the third rule in Def. 5, we have $\tau_1.\mathbf{a}.\tau_3 \in \mathcal{L}(\psi, \theta)$.

Next, we discuss restrictions that this invariance imposes on formulas. This will motivate a syntactic fragment that allows only sliceable formulas to be written.

4.2 Restrictions on the structure of formulas

Not all $\mathsf{FO\text{-}LTL}_f$ formulas are sliceable. We discuss restrictions that the notion of slicing invariance places on $\mathsf{FO\text{-}LTL}_f$ formulas. We use Fig. 1 to illustrate these restrictions. It gives an example trace where shaded states are not relevant to a formula ψ over $\{x\}$ and to the valuation $[x = b]$.
Top-level quantification: Consider the following formula with *embedded quantifiers* $\exists x : \Box(\mathtt{f}(x) \rightarrow \exists y : \Diamond \mathtt{h}(x, y))$. It is not possible to rewrite this so that the $\exists y$ appears at the top level. Our example trace satisfies the property, but we cannot use slicing to capture this [1]. In the definition of slicing in Sec. 3 *all* variables must be used for slicing. To check this property for $x = b$ it would be necessary to include every $\mathtt{h}(b, y)$ event for an arbitrary number of ys, requiring an arbitrary number of quantifications. Therefore, we consider only $\mathsf{FO\text{-}LTL}_f$ formulas that have quantification at the top level.

[1] Here we refer to the notion of slicing in Sec. 3. A more general notion of slicing involves free variables [3], which could be used for y. It is future work to consider this generalisation.

Starting at the start: Consider the formula $\mathtt{f}(x) \vee \Diamond \mathtt{k}(x)$ as in Fig. 1. The trace does not satisfy this formula for $x = b$. The first event is not $\mathtt{f}(b)$ and there is no $\mathtt{k}(b)$. However, in the slicing approach, the first event for the $[x = b]$-slice is $\mathtt{f}(b)$ and this would be accepted. The formula $\neg\mathtt{f}(x) \vee \Diamond \mathtt{k}(x)$ would have similar (symmetrical) discrepancy. This means that we cannot allow events (in positive or negative form) at the top level of a sliceable formula. Furthermore, we cannot allow $\mathcal{U}^\circ$ formulas at the top level, as it is not possible to tell whether it should be evaluated from the first or second event in a trace slice, as illustrated by the formula $\neg\mathtt{f}(x)\, \mathcal{U}^\circ\, \mathtt{g}(x)$ for the trace $\mathtt{g}(a).\mathtt{g}(a).\mathtt{f}(b).\mathtt{g}(b)$.

Never saying next: Consider the formula $\Box(\mathtt{f}(x) \rightarrow \bigcirc \mathtt{g}(x))$ on the trace in Fig. 1. For $x = b$ we have $\mathtt{g}(a)$ after a $\mathtt{f}(b)$; however, in the $[x = b]$-slice this is removed. This shows that evaluation differs after slicing. Formulas must therefore be *next-free*. This *next-freeness* extends to the left side of $\mathcal{U}$ and $\mathcal{U}^\circ$ formulas in general. Consider the formula $\Box(\mathtt{f}(x) \vee \mathtt{g}(x)) = (\mathtt{f}(x) \vee \mathtt{g}(x))\, \mathcal{U}\, \mathit{false}$ as in Fig. 1. The trace does not satisfy the formula; however, for the $[x = b]$-slice it would as the non-relevant events violating $\mathtt{f}(x) \vee \mathtt{g}(x)$ are removed. This shows that the left side of any $\mathcal{U}$ or $\mathcal{U}^\circ$ formula must evaluate to *true* for non-relevant events.

Never saying never: Consider the formula $\Diamond(\neg\mathtt{f}(x) \wedge \neg\mathtt{g}(x)) = \mathit{true}\, \mathcal{U}\, (\neg\mathtt{f}(x) \wedge \neg\mathtt{g}(x))$ as in Fig. 1. Here the obligation $(\neg\mathtt{f}(x) \wedge \neg\mathtt{g}(x))$ is never satisfied in the $[x = b]$-slice but it is in the full trace. This shows that the right side of an $\mathcal{U}$ or $\mathcal{U}^\circ$ formula must evaluate to *false* for non-relevant events.

Symmetry: Note that there is a symmetry between the restrictions on the left and right of $\mathcal{U}$ and $\mathcal{U}^\circ$ formulas. This is due to the following identity $\psi_2\, \mathcal{U}\, \psi_3 = \neg(\neg\psi_3\, \mathcal{U}\, \neg\psi_2)$ which can turn a restriction on the left to one on the right.

4.3 A syntactic fragment

We introduce a syntactic fragment $\mathcal{F}$ of $\mathsf{FO\text{-}LTL}_f$ that incorporates the restrictions discussed above. The fragment consists of formulas $Q_1 x_1 : \ldots Q_n x_n : \psi_T$ for zero or more quantifications $Q_i x_i$, with $Q_i = \forall$ or $\exists$, and quantifier-free ψ_T inductively defined as :

$$\begin{aligned}
\psi_T &= \psi_L\, \mathcal{U}\, \psi_R \mid \psi_T \vee \psi_T \mid \psi_T \wedge \psi_T \\
\psi_L &= \mathit{true} \mid \psi_L \vee \psi_U \mid \psi_L \wedge \psi_L \mid \neg \mathbf{a} \\
\psi_R &= \mathit{false} \mid \psi_R \wedge \psi_U \mid \psi_R \vee \psi_R \mid \mathbf{a} \\
\psi_U &= \psi_L\, \mathcal{U}^\circ\, \psi_R \mid \psi_L\, \mathcal{U}\, \psi_R \mid \psi_U \vee \psi_U \mid \psi_U \wedge \psi_U
\end{aligned}$$

where **a** is an event. This syntax captures the restrictions discussed above: the restricted form of ψ_T captures the restrictions on 'start', and the restrictions on the left and right of $\mathcal{U}$ and $\mathcal{U}^\circ$ formulas are captured by the restricted forms of ψ_L and ψ_R respectively.

Not all identities in $\mathsf{FO\text{-}LTL}_f$ can be expressed in the fragment $\mathcal{F}$. For example, the definition of the next modality $\bigcirc\phi = \mathit{false}\, \mathcal{U}^\circ\, \phi$ is not in $\mathcal{F}$, as *false* cannot occur by itself on the left side of a $\mathcal{U}^\circ$ formula. The identity $\Diamond\psi_R = \mathit{true}\, \mathcal{U}\, \psi_R$ however, for quantifier-free ψ_R, is expressed in $\mathcal{F}$, but $\Box\phi = \phi\, \mathcal{U}\, \mathit{false}$ is not in general in $\mathcal{F}$ because of the restrictions placed on the left side of $\mathcal{U}^\circ$. However, $\Box(\mathbf{a} \rightarrow \psi_R)$ is expressible in $\mathcal{F}$ as $\neg\mathbf{a} \vee \psi_R$ is an allowed left formula for $\mathcal{U}$.

We show that formulas in $\mathcal{F}$ are *sliceable*. Lemma 2 first shows that the structure of formulas in $\mathcal{F}$ allows non-relevant events to be added and removed from a trace.

Lemma 2. *For any formula $\psi \in \mathcal{F}$ with free variables X, traces τ_1 and τ_2, valuations θ over X, events $\mathbf{a} \notin \mathsf{relevant}(\psi, \theta)$ and indices i and j:*

> ***Case 1.*** *If ψ is in ψ_L, ψ_R or ψ_U and $(\tau_1.\mathbf{a}.\tau_2)_i \in \mathsf{relevant}(\psi, \theta)$ then $\tau_1.\mathbf{a}.\tau_2, \theta, i \models \psi \Leftrightarrow \tau_1.\tau_2, \theta, j \models \psi$ where $j = i$ if $i < |\tau_1|$ or $j = i - 1$ otherwise.*
> ***Case 2.*** *If ψ is in ψ_L then $\tau_1.\mathbf{a}.\tau_2, \theta, |\tau_1| \models \psi$ i.e. ψ holds at $\mathbf{a}$*
> ***Case 3.*** *If ψ is in ψ_R then $\tau_1.\mathbf{a}.\tau_2, \theta, |\tau_1| \not\models \psi$ i.e. ψ does not hold at $\mathbf{a}$*
> ***Case 4.*** *If ψ is in ψ_T then $\tau_1.\mathbf{a}.\tau_2, \theta, 0 \models \psi \Leftrightarrow \tau_1.\tau_2, \theta, 0 \models \psi$*

Proof. By simultaneous structural induction on ψ, under the assumption that the properties hold for appropriate subformulas.

Case 1. If $i \neq |\tau_1|$ then $(\tau_1.\mathbf{a}.\tau_2)_i = (\tau_1.\tau_2)_j$ and $i = |\tau_1|$ is not allowed as the event at $(\tau_1.\mathbf{a}.\tau_2)_i$ must be relevant. The base cases *true*, *false*, $\mathbf{a}$ and $\neg\mathbf{a}$ hold trivially. The conjunctive and disjunctive cases follow from the induction hypothesis. The interesting cases are $\mathcal{U}$ and $\mathcal{U}^\circ$, we consider $\mathcal{U}$ and then describe the extension to $\mathcal{U}^\circ$.

For the $\Rightarrow$ direction assume $\tau_1.\mathbf{a}.\tau_2, \theta, i \models \psi_L \, \mathcal{U} \, \psi_R$ and therefore there must exist a $k \geq i$ such that $\tau_1.\mathbf{a}.\tau_2, \theta, k \models \psi_R$. By the induction hypothesis $\tau_1.\tau_2, \theta, l \models \psi_R$ where l depends on the location of $\mathbf{a}$, note that the induction hypothesis can be used as by case 3, the event at k is relevant. By the assumption, ψ_L holds at all points $m \geq i$ and $m < l$ in $\tau_1.\mathbf{a}.\tau_2$ and so by the induction hypothesis and case 2 (when the point is not relevant) it holds in all such points in $\tau_1.\tau_2$. Therefore, $\tau_1.\tau_2, \theta, j \models \psi_L \, \mathcal{U} \, \psi_R$.

The $\Leftarrow$ direction is similar. Again, we can assume that $\tau_1.\tau_2, \theta, k \models \psi_R$ for some $k \geq i$ and therefore, by the induction hypothesis, $\tau_1.\mathbf{a}.\tau_2, \theta, l \models \psi_R$. Also we argue by the assumption, the induction hypothesis and case 2 that ψ_L holds for all points i to l, including the new $|\tau_1|$ if included in the range. Therefore, $\tau_1.\mathbf{a}.\tau_2, \theta, i \models \psi_L \, \mathcal{U} \, \psi_R$.

For the $\mathcal{U}^\circ$ case the proof is similar but we reason about ψ_R being satisfied at a point k *strictly greater* than i. This relies on τ_i being relevant as this prevents $i = |\tau_1|$, which is necessary for the $\Rightarrow$ direction. If $i = |\tau_1|$ and $k = |\tau_1| + 1$ then due to the semantics of $\mathcal{U}^\circ$ we can no longer argue that $\tau_1.\tau_2, \theta, l \models \psi_R$ implies that $\tau_1.\tau_2, \theta, j \models \psi_L \, \mathcal{U}^\circ \, \psi_R$ as we would have $l = j$.

Case 2. If $\psi = \mathit{true}$ then this holds trivially. If $\psi = \neg\mathbf{b}$ then this will be satisfied by all non-relevant $\mathbf{a}$. Both the conjunctive and disjunctive cases follows from the inductive hypothesis as in the $\wedge$ case both parts are in ψ_L and in the $\vee$ case at least one is.

Case 3. If $\psi = \mathit{false}$ then this holds trivially. For $\psi = \mathbf{b}$, $\mathbf{b} \neq \mathbf{a}$ as $\mathbf{a}$ is not relevant. Again the conjunctive and disjunctive cases follow from the inductive hypothesis.

Case 4. The conjunctive and disjunctive cases follow from the inductive hypothesis. The $\mathcal{U}$ case is the same as the argument above for case 1 where $i = 0$ and the condition that τ_i is relevant can be dropped as this is not used.

Finally, any formula that can be expressed in $\mathcal{F}$ is sliceable:

Theorem 1. *All formulas in $\mathcal{F}$ are sliceable.*

Proof. (Sketch) Any $\psi \in \mathcal{F}$ is slicing invariant. This follows from Lemma 2 by induction on the length of τ using Def. 5 (similar to the proof of Lemma 1).

$$\frac{\overline{x} = \overline{y}}{\mathsf{e}(\overline{y}) \xrightarrow{\mathsf{e}(\overline{y})} true} \quad \frac{\overline{x} \neq \overline{y}}{\mathsf{e}(\overline{y}) \xrightarrow{\mathsf{e}(\overline{y})} false} \quad \frac{\psi \xrightarrow{\mathsf{e}(\overline{y})} \psi'}{\neg\psi \xrightarrow{\mathsf{e}(\overline{y})} \neg\psi'} \quad \frac{\psi_1 \xrightarrow{\mathsf{e}(\overline{y})} \psi_1' \quad \psi_2 \xrightarrow{\mathsf{e}(\overline{y})} \psi_2'}{\psi_1 \wedge \psi_2 \xrightarrow{\mathsf{e}(\overline{y})} \psi_1' \wedge \psi_2'}$$

$$\frac{\psi_1 \xrightarrow{\mathsf{e}(\overline{y})} \psi_1' \quad \psi_2 \xrightarrow{\mathsf{e}(\overline{y})} \psi_2'}{\psi_1 \vee \psi 2 \xrightarrow{\mathsf{e}(\overline{y})} \psi_1' \vee \psi_2'} \quad \frac{}{\psi_1 \, \mathcal{U}^\circ \, \psi_2 \xrightarrow{\mathsf{e}(\overline{y})} \psi_2 \vee (\psi_1 \wedge (\psi_1 \, \mathcal{U}^\circ \, \psi_2))}$$

$$\begin{array}{ll} \texttt{accept}(true) = true & \texttt{accept}(\neg\psi) = \text{not } \texttt{accept}(\psi) \\ \texttt{accept}(\mathbf{a}) = false & \texttt{accept}(\psi_1 \, \mathcal{U}^\circ \, \psi_2) = \\ \texttt{accept}(\psi_1 \wedge \psi_2) = \texttt{accept}(\psi_1) \text{ and } \texttt{accept}(\psi_2) & \quad true \quad \text{if } \psi_2 = false \\ \texttt{accept}(\psi_1 \vee \psi_2) = \texttt{accept}(\psi_1) \text{ or } \texttt{accept}(\psi_2) & \quad false \quad \text{otherwise} \end{array}$$

Fig. 2. The progression and acceptance rules.

5 From Temporal Logic to Automata

In this section, we introduce a translation from formulas in $\mathcal{F}$ to the slicing-based formalism QEA. This is based on the notion of *progression* and a normal form that ensures a finite number of syntactically different formulas resulting from progression.

5.1 Progression

Fig. 2 gives the progression and acceptance rules for $\mathsf{FO\text{-}LTL}_f$ formulas. The progression rules show how formulas are rewritten, note that these convert formulas in $\mathcal{F}$ to formulas not necessarily in $\mathcal{F}$. The acceptance rules capture whether a formula is currently accepting. Firstly, we note that progression preserves the semantics of $\mathsf{FO\text{-}LTL}_f$:

Lemma 3. *For every $\mathsf{FO\text{-}LTL}_f$ formula ψ, valuation θ and trace τ we have*

$$\tau, \theta, 0 \models \psi \quad \Leftrightarrow \quad \tau, \theta, |\tau| \models \psi' \text{ for } \psi \xrightarrow{\tau\downarrow_\theta} \psi'$$

Proof. (Sketch) By induction on the structures of τ then ψ, noting that the progression rules follow $\models$ (in Def 1). For a similar proof, see Lemmas 3 and 4 in [6].

Secondly, the acceptance rules capture the desired behaviour i.e. what the verdict would be if the trace terminated at this point. Note that the rule for $\mathcal{U}^\circ$ reflects the notion of outstanding obligations alongside the finite-trace interpretation of $\Box$:

Lemma 4. *For every $\psi \in \mathcal{F}$ and valuation θ*

$$\epsilon, \theta, 0 \models \psi \Leftrightarrow \tau, \theta, |\tau| \models \psi \Leftrightarrow \texttt{accept}(\theta(\psi))$$

Proof. (Sketch) By induction on the structure of ψ, we show that `accept` respects the semantics $\models$.

5.2 A normal form

We give a normal form that gives an upper bound for the number of progression steps to syntactically different formulas. This approach was inspired by [18] and a similar result has recently been established in [19].

The normal form is given by the following rewrite rules.

$$
\begin{array}{llllll}
\mathit{true} \wedge \psi & \rightarrow \psi & \neg\neg\psi & \rightarrow \psi & \\
\psi \wedge \mathit{true} & \rightarrow \psi & \psi_1 \wedge \psi_2 & \rightarrow \psi_1 & \text{if } \psi_1 = \psi_2 \\
\neg\mathit{true} \vee \psi & \rightarrow \psi & \psi_1 \vee \psi_2 & \rightarrow \psi_1 & \text{if } \psi_1 = \psi_2 \\
\psi \vee \neg\mathit{true} & \rightarrow \psi & (\bigvee_i \psi_i) \wedge (\bigwedge_j \psi_j) & \rightarrow \bigvee_i(\psi_i \wedge (\bigwedge_j \psi_j)) &
\end{array}
$$

A formula is in normal form if none of these rewrite rules can be applied to any of its subformulas. We write $\mathtt{nf}(\psi)$ for the normal form of ψ. If $\psi \in \mathcal{F}$ then $\mathtt{nf}(\psi)$ is a disjunction of conjunctions where each conjunct is either an event or a *temporal formula* i.e. a $\mathcal{U}^\circ$ formula. We first show:

Lemma 5. *For $\psi \in \mathcal{F}$, let $\mathcal{T}$ be the temporal subformulas of ψ, every formula ψ' such that $\psi \xrightarrow{\tau} \psi'$ for any trace τ, has temporal subformulas $\mathcal{T}' \subseteq \mathcal{T}$.*

Proof. The only progression rule dealing with temporal formulas is the $\mathcal{U}^\circ$ rule. This copies the temporal formula. Therefore, no new temporal formulas are created.

Next we show that there can be no infinite progression sequences without repeating formulas syntactically equivalent up to normal form.

Lemma 6. *Any formula $\psi \in \mathcal{F}$ has a finite number of formulas ψ' such that $\psi \xrightarrow{\tau} \psi'$ for any trace τ.*

Proof. Let $\mathcal{P}(\psi)$ be the set of formulas in normal form that can be built from boolean combinations of events in ψ and formulas in $\mathcal{T}$, the temporal subformulas of ψ. The set $\mathcal{P}(\psi)$ is finite as there are a finite number of events and $\mathcal{T}$ is finite. From Lemma 5, ψ', we have $\psi' \in \mathcal{P}(\psi)$ and therefore there are a finite number of such ψ'.

Furthermore, $\mathcal{P}(\psi)$ is bounded by $2^{|\psi|}$ (see the result in [19]), giving an upper bound on such a sequence.

5.3 Progression-based translation

We now introduce a translation based on progression. We begin by introducing a translation from quantifier-free formulas to state machines.

Definition 6 (Quantifier-free translation). *Given a quantifier-free formula $\psi \in \mathcal{F}$, let $\mathtt{translate}(\psi) = \langle Q, q_0, \mathcal{A}, \delta, F \rangle$ be the automaton such that*

$$
\begin{array}{ll}
\mathcal{A} = \mathtt{events}(\psi) & Q = \{q_0\} \cup \{\mathtt{nf}(\psi') \mid \exists \tau : \psi \xrightarrow{\tau} \psi'\} \\
q_0 = \mathtt{nf}(\psi) & \delta(\psi', \mathbf{a}) = \mathtt{nf}(\psi'') \textit{ where } \psi' \in Q, \mathbf{a} \in \mathcal{A} \textit{ and } \psi' \xrightarrow{\mathbf{a}} \psi'' \\
& F = \{\psi' \in Q \mid \mathtt{accept}(\psi')\}
\end{array}
$$

It is important that the constructed state machine is finite, so that this translation step terminates. This follows from Lemma 6, as there are only a finite number of syntactically distinct ψ' such that $\psi \xrightarrow{\tau} \psi'$. This also puts an upper bound of $2^{|\psi|}$ on the number of states of the automata.

There exists a simple procedure for building this state machine that begins with a set of states S containing the initial formula/state and the set of events $\mathcal{A}$ and considers all possible progressions from states in S for events in $\mathcal{A}$, producing a new set of reachable states S'. This is then repeated for $S' \backslash S$.

This automata translation captures the progression semantics directly:

Lemma 7. *For every quantifier-free formula ψ and trace τ*

$$\psi \xrightarrow{\tau} \psi' \quad \Leftrightarrow \quad \delta(\psi, \tau) = \psi'$$

where δ is the transition relation of $\texttt{translate}(\psi)$

Proof. (Sketch) By induction on the structure of τ. Note that δ is defined directly in terms of progression.

We then define the full translation from quantified temporal formulas in $\mathcal{F}$ to QEA.

Definition 7 (Translation). *Let* $\texttt{translate}(\phi) = \langle \Lambda(X), \mathcal{E} \rangle$ *such that for* $\phi = Q_1 x_1 : \ldots Q_n x_n : \psi$, $\Lambda(X) = Q_1 x_1 : \ldots Q_n x_n$ *and* $\mathcal{E} = \texttt{translate}(\psi)$.

5.4 An example of the translation

Let us consider the translation of the standard HasNext temporal formula that will be described in Sec. 6. In the following, we use $\mathbf{h} = \texttt{hasNext}(i)$ and $\mathbf{n} = \texttt{next}(i)$. The formula for the property is given as follows where we rewrite $\mathcal{U}$, $\Box$ and $\rightarrow$.

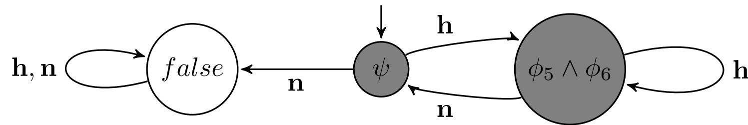
$$\psi = (\mathbf{h} \vee (\mathbf{n} \wedge (\neg \mathbf{n}\, \mathcal{U}^\circ\, \mathbf{h}))) \wedge (\neg \mathbf{n} \vee (\neg \mathbf{n}\, \mathcal{U}^\circ\, \mathbf{h})) \wedge ((\neg \mathbf{n} \vee (\neg \mathbf{n}\, \mathcal{U}^\circ\, \mathbf{h}))\, \mathcal{U}^\circ\, \mathit{false})$$

The following shows the *normal form* of rewriting each subformula of ψ with respect to the events $\mathbf{h}$ and $\mathbf{n}$.

$$\begin{aligned}
\phi_1 &= \mathbf{n} && \xrightarrow{\mathbf{h}} \ false \\
& && \xrightarrow{\mathbf{n}} \ true \\
\phi_2 &= \mathbf{h} && \xrightarrow{\mathbf{h}} \ true \\
& && \xrightarrow{\mathbf{n}} \ false \\
\phi_3 &= \neg \phi_1\, \mathcal{U}^\circ\, \phi_2 && \xrightarrow{\mathbf{h},\mathbf{n}} \phi_4
\end{aligned}$$

$$\begin{aligned}
\phi_4 &= \phi_2 \vee (\neg \phi_1 \wedge \phi_3) && \xrightarrow{\mathbf{h}} \ true \\
& && \xrightarrow{\mathbf{n}} \ false \\
\phi_5 &= \neg \phi_1 \vee \phi_3 && \xrightarrow{\mathbf{h}} \ true \\
& && \xrightarrow{\mathbf{n}} \ \phi_4 \\
\phi_6 &= \phi_5\, \mathcal{U}^\circ\, false && \xrightarrow{\mathbf{h},\mathbf{n}} \phi_5 \wedge \phi_6 \\
\psi &= \phi_4 \wedge \phi_6 && \xrightarrow{\mathbf{h}} \ \phi_5 \wedge \phi_6 \\
& && \xrightarrow{\mathbf{n}} \ false
\end{aligned}$$

From this we can observe three states: ψ, *false* and $\phi_5 \wedge \phi_6$. Observe that $\phi_5 \wedge \phi_6 \xrightarrow{\mathbf{n}} (\phi_4 \wedge \phi_5 \wedge \phi_6) = \psi$. The final states are given by $\texttt{accept}$ where $\texttt{accept}(\mathit{false}) = \mathit{false}$ and $\texttt{accept}(\psi)$ and $\texttt{accept}(\phi_5 \wedge \phi_6)$ are *true* due to their top level $\mathcal{U}^\circ$ operators. This gives the following automaton.



5.5 Correctness of translation

We show that the translated QEA is trace-equivalent to the original formula. We first establish the unquantified case.

Lemma 8. *For any unquantified $\psi \in \mathcal{F}$ with free variables X, trace τ, and valuation θ over X*

$$\tau, \theta \models \psi \quad \Leftrightarrow \quad \tau \downarrow_\theta \in \mathcal{L}(\texttt{translate}(\psi))$$

Proof. As $\psi \in \mathcal{F}$ is sliceable (by Theorem 1) this can be rewritten as

$$\tau \downarrow_\theta, \theta \models \psi \quad \Leftrightarrow \quad \tau \downarrow_\theta \in \mathcal{L}(\texttt{translate}(\psi)) \tag{1}$$

By definition $\mathcal{L}(\texttt{translate}(\psi)) = \{\tau \mid \delta(\psi, \tau) = \psi' \text{ and } \texttt{accepts}(\psi')\}$. Therefore, by Lemmas 7 and 4 the right side of (1) can be rewritten

$$\psi \xrightarrow{\tau\downarrow_\theta} \psi' \text{ and } \tau \downarrow_\theta, \theta, |\tau \downarrow_\theta| \models \psi' \tag{2}$$

and by Lemma 3, (2) can be rewritten as $\tau \downarrow_\theta, \theta \models \psi$, showing the equivalence holds.

This can be used to show the correctness in the quantified case.

Theorem 2. *For every trace τ and a formula $\phi \in \mathcal{F}$*

$$\tau \models \phi \quad \Leftrightarrow \quad \tau \models_{[]}^{\mathcal{P}(X)} \Lambda(X)$$

where $\texttt{translate}(\phi) = \langle \Lambda(X), \mathcal{P}(X) \rangle$.

Proof. (Sketch) By structural induction on the quantification structure of ϕ and Lemma 8.

6 The fragment $\mathcal{F}$ as a trace specification language

Are syntactic fragments of $\mathsf{FO\text{-}LTL}_f$ that respect a notion of sliceability expressive enough to be considered as practically useful trace specification languages? We consider how we may answer this question for the syntactic fragment $\mathcal{F}$. However, assessing the (practically useful) expressiveness of a specification language for runtime verification is currently not easy and there are no accepted methods, or accepted corpora of cases or specification patterns likely to occur in real runtime verificaion applications.

Common patterns. In 1999 Dwyer et al. conducted a survey of common finite-state properties used for *model-checking* in industry and academia [11]. We consider whether some of the more common properties are expressible in $\mathcal{F}$.

A common pattern is that of *response* written $\Box(P \rightarrow \Diamond Q)$. This can be rewritten to $(\neg P \vee (\mathit{true}\, \mathcal{U}\, Q))\, \mathcal{U}\, \mathit{false}$, which is a formula expressed in the fragment. Another pattern in the collection is an *absence* property stating that P does not occur before R. We show how this can be rewritten into a formula expressed in the fragment.

$$(\Diamond R \rightarrow \neg P\, \mathcal{U}\, R) = (\neg(\mathit{true}\, \mathcal{U}\, R) \vee (\neg P\, \mathcal{U}\, R)) = (\neg R\, \mathcal{U}\, \mathit{false}) \vee (\neg P\, \mathcal{U}\, R)$$

A more complex absence property that may appear to be outside of the fragment at first is that P does not occur between Q and R, written as $\Box((Q \wedge \bigcirc\Diamond R) \rightarrow (\neg P \wedge$

$\bigcirc(\neg P\,\mathcal{U}\,R)))$. This can be written using the identities such as $\bigcirc\Diamond R = \mathit{true}\,\mathcal{U}^{\circ}\,R$ to be $(\neg Q \vee (\neg R\,\mathcal{U}^{\circ}\,\mathit{false}) \vee (\neg P \wedge (\neg P\,\mathcal{U}^{\circ}\,R)))\,\mathcal{U}\,\mathit{false}$.

A set of patterns from this study that cannot be expressed in $\mathcal{F}$ are the general *universality* patterns $\Box P$. However, specific cases of these may be expressible.

This is but a small sample, and only for common model-checking properties, but the technique is clear.

Specifications in the literature. There are specifications commonly occurring as examples in RV literature that belong to the fragment $\mathcal{F}$, we give some of these here.

HasNext. For every iterator i the first event is $\mathtt{hasNext}(i)$ and whenever a $\mathtt{next}(i)$ event occurs there is not another $\mathtt{next}(i)$ event until there has been a $\mathtt{hasNext}(i)$ event. This can be captured by the following formula in $\mathcal{F}$.

$$\forall i : (\neg\mathtt{next}(i)\,\mathcal{U}\,\mathtt{hasNext}(i)) \wedge \Box(\mathtt{next}(i) \rightarrow (\neg\mathtt{next}(i)\,\mathcal{U}^{\circ}\,\mathtt{hasNext}(i)))$$

UnsafeMapIter. For every map m, collection c and iterator i, whenever c is created from m and i is created from c, after m is updated i should not be used.

$$\forall m : \forall c : \forall i : \Box(\mathtt{create}(m,c) \rightarrow \Box(\mathtt{iterator}(c,i) \rightarrow \Box(\mathtt{update}(m) \rightarrow \Box\neg\mathtt{use}(i))))$$

CallNesting. The call-nesting property given to motivate the slicing approach earlier can also be specified in $\mathcal{F}$ as follows.

$$\begin{array}{l}\forall m_1 : \forall m_2 : (\neg\mathtt{return}(m_1)\,\mathcal{U}\,\mathtt{call}(m_1)) \wedge (\neg\mathtt{return}(m_2)\,\mathcal{U}\,\mathtt{call}(m_2)) \wedge \\ \quad \Box(\mathtt{call}(m_1) \rightarrow (\neg\mathtt{call}(m_1)\,\mathcal{U}\,\mathtt{return}(m_1))) \wedge \Box(\mathtt{call}(m_1) \rightarrow \\ \quad (\mathtt{call}(m_2) \rightarrow ((\neg\mathtt{return}(m_2) \wedge \neg\mathtt{call}(m_2))\,\mathcal{U}\,\mathtt{return}(m_2)))\,\mathcal{U}\,\mathtt{return}(m_1))\end{array}$$

Note how the formula requires many parts to capture the different paths through the previously defined automaton. This demonstrates the differing usability of the two approaches. This study could be extended by considering the slicing-based specifications for the `Java` JDK given in [13]. We expect all slicing-based specifications from work with JAVAMOP that use a regular propositional language to be expressible in $\mathcal{F}$.

7 Related Work

This work aims to connect approaches based on parametric trace slicing with those based on first-order temporal logic. We give an overview of related work in each area. We have not considered the rule-based system approach [4], where some work has linked the expressiveness of rule systems with (propositional) temporal logic [4].

Slicing. Arguably, the first system to use trace-slicing was tracematches [2], but the paper did not use this terminology, and the suffix-based matching meant that the authors did not need to solve the main technical difficulty in slicing i.e. dealing with partial bindings. The JAVAMOP system [15] has made the slicing approach popular with its highly efficient implementation. The QEA formalism [3, 16] and associated MARQ tool [17] were inspired by JAVAMOP. The notion of slicing presented here is compatible with that used in JAVAMOP. Note that the combination of slicing and propositional LTL used in JAVAMOP does not correspond to a first-order temporal logic. For example, the JAVAMOP property $\Lambda x.\Box(\mathtt{f}(x) \rightarrow \bigcirc\mathtt{g}(x))$ does not have the standard first-order temporal logic interpretation as discussed in Sec. 4.

First-order temporal logics in RV. Most approaches that add first-order reasoning to LTL for runtime verification use similar concepts, with the main difference typically being the *domain of quantification*. An early work extending LTL by parameters by Stolz and Bodden [20] makes bindings locally in a PROLOG-style. Bauer et al. [7] have proposed a variant of first-order LTL where quantification is restricted to the values known at a single point in time. Decker et al. [10] introduce the notion of *temporal data logic*, an extension of temporal logic with first-order theories. Monpoly [5] constructs monitors for a safety-fragment of metric first-order temporal logic where temporal operators are augmented with intervals. Yoshi et al. [14] introduce a parallel monitoring approach for first-order LTL extended with second-order numerical constraints. In principle it would be possible to consider restricting all of these logics syntactically so that they could be sliceable. However, in some cases the syntactic fragment may not be useful.

8 Conclusion

The aim of this work is to explore the relationship between first-order temporal logic and parametric trace slicing. We have introduced a technique based on identifying syntactic fragments of temporal logics which respect trace slicing, and defined one such fragment of $\textsf{FO-LTL}_f$. From this fragment, we have shown how we may construct automata with a slicing interpretation i.e. QEA.

The notion of trace slicing, and hence sliceability, used in this work is more restrictive than that used in [3] as it requires all variables to participate in slicing. We briefly discuss possible future work that could lead to different fragments, perhaps of more expressive temporal logics:

Embedded quantification. There are cases when embedded quantification can be necessary in specification e.g. $\forall x : \forall t_1 : \Box(\texttt{start}(x, t_1) \rightarrow \exists t_2 : \texttt{stop}(x, t_2))$. Here the existential quantification of t_2 cannot be lifted outside of the scope of $\Box$. However, embedded quantification is supported in QEA using *free variables*.

Predicates and functions. First-order logics usually include predicates and functions. A simple extension would allow predicates on quantified values of the form $\forall x : p(x) \rightarrow \psi$ and $\exists x : p(x) \wedge \psi$. The translation to the slicing setting would be straightforward as QEA support global guards of this form. A more general incorporation of predicates and functions would be supported by the guard and assignments on transitions in the QEA formalism; this would be most appropriate alongside embedded quantification. Predicates and functions require interpretation. These interpretations could be supplied on a built-in or ad-hoc basis or taken from a formal theory, as is done in [10].

Translation from QEA **to temporal logic.** Even the restricted form of QEA used in this paper are more expressive than the temporal logic, as it inherits the star-freeness of standard LTL. For example, the language ‘there are an even number of a(x) events’ cannot be expressed in the logic, but can be expressed as a QEA. Therefore, to translate QEA to temporal logic a more expressive temporal logic is required.

Whilst the fragment $\mathcal{F}$ introduced here does not cover all specifications that might be written in a slicing framework, we have considered how we may assess its practical expressiveness and provided a technique for translating formulas in $\mathcal{F}$ to QEA.

We believe the goal of this work, seeking methods for unifying existing specification languages, is of considerable importance in runtime verification in allowing us to improve the comparability and interoperability of tools.

References

1. CSRV 2014. `http://rv2014.imag.fr/monitoring-competition`.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, 2005.
3. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pp. 68–84, 2012.
4. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. *J Logic Computation*, 20(3):675–706, 2010.
5. D. Basin, M. Harvan, F. Klaedtke, and E. Zlinescu. Monpoly: Monitoring usage-control policies. In *Runtime Verification*, vol. 7186 of *Lecture Notes in Computer Science*, pp. 360–364. Springer Berlin Heidelberg, 2012.
6. A. Bauer and Y. Falcone. Decentralised LTL monitoring. *CoRR*, abs/1111.5133, 2011.
7. A. Bauer, J.-C. Kster, and G. Vegliach. The ins and outs of first-order runtime verification. *Formal Methods in System Design*, pp. 1–31, 2015.
8. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, 2010.
9. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, vol. 5505 of *LNCS*, pp. 246–261, 2009.
10. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014.*, pp. 341–356, 2014.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pp. 411–420. ACM, 1999.
12. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
13. C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, 2012.
14. R. Medhat, Y. Joshi, B. Bonakdarpour, and S. Fischmeister. Parallelized runtime verification of first-order LTL specifications. Technical report, University of Waterloo, 2014.
15. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pp. 1–41, 2011.
16. G. Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
17. G. Reger, H. C. Cruz, and D. Rydeheard. MARQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
18. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.
19. Y. Shen, J. Li, Z. Wang, T. Su, B. Fang, G. Pu, and W. Liu. Runtime verification by convergent formula progression. *APSEC 2014*.
20. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, vol. 144(4) of *ENTCS*, pp. 109–124. Elsevier, 2006.