Computer Says No: Verdict Explainability for Runtime Monitors using a Local Proof System[⋆,⋆⋆]

Adrian Francalanza[a,*], Clare Cini[b]

[a] *CS@ICT, University of Malta, Msida, Malta, MSD2080.*
[b] *Ricston Ltd., G.F. Agius De Soldanis Street, Birkirkara, Malta, BKR4850.*

Abstract

Monitors in Runtime Verification are often constructed as black boxes: they provide verdicts on whether a property is satisfied or violated by the executing system under scrutiny, without much explanation as to why this is the case. In the best of cases, monitors might also return the trace observed, still leaving it up to the user to figure out the logic employed to reach the declared verdict from this trace. In this paper, we propose a local proof system for Linear Temporal Logic—a popular logic used in Runtime Verification—formalising the symbolic deductions within the constraints of Runtime Verification. We prove novel soundness and partial completeness results for this proof system with respect to the original semantics of the logic. Crucially, we show how such a deductive system can be used as a realistic basis for constructing online runtime monitors that provide explanations for their verdicts; we also show the resulting monitor algorithms to satisfy pleasing correctness criteria identified by other works, such as the decidability and incrementality of the analysis and the irrevocability of verdicts. Finally, we relate the expressiveness of the Linear Temporal Logic proof system to existing symbolic analysis techniques used in Runtime Verification.

Keywords: Runtime Verification, Explainability, Interpretability, Linear Temporal Logic, Proof Systems, Monitoring, Correct Monitor Synthesis

[⋆] The research was supported by the University of Malta Research Fund project "A Theory of Monitors" CPSRP05-04, the Icelandic Research Fund RANNIS project "Theoretical Foundations of Monitorability" TheoFoMon:163406-051 and by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 "BehAPI: Behavioural APIs".

[⋆⋆] A preliminary version of this work appeared in [28]. This article includes expanded explanations, more detailed examples, and the complete proofs for the stated theorems. Section 4 has been substantially extended and includes additional results and motivations. We have also augmented the related work section to include recent literature.

[*] Corresponding author

1. Introduction

Runtime verification (RV) [51, 12] is a lightweight verification technique that checks whether the current execution of the system under scrutiny satisfies or violates a given correctness property. It has its origins in model checking, as a more scalable (yet still formal) approach to program verification: it was recognised early on that, for certain properties, it is far cheaper to search for a (witness) execution path that disproves the property [52, 50]. In an RV setup, correctness properties are synthesised as *monitors*, executables that analyse a system execution *incrementally* to report *irrevocable* verdicts that relate the observed execution with the monitored property [12, 5]. The technique has been applied to a variety of settings, from the healthcare sector to financial services [60]. RV has proved to be particularly useful to verify *open* distributed systems where a subset of the participating processes might not necessarily statically verified [39]. The technique has been employed in conjunction to other verification techniques [8, 30, 48, 46, 56, 1, 24], in a variety of configurations that include centralised and choreographed arrangements [38, 16, 55, 20, 9].

Although beneficial, it can be argued that having a monitor that simply returns a verdict is of little use to an engineer who wants to understand the *cause* of the violation or satisfaction. Such an understanding is crucial for either resolving the errors causing the violation or improving system performance (while preserving the good behaviour [21, 29]). In principle, one could mitigate this by also returning the trace observed[1] leading to the verdict. But this still leaves the engineer with a lot of work to do. For instance, it is hard to infer which sub-property in a conjunction was violated (dually, which sub-property in a disjunction was satisfied) or at which iteration an invariant property was violated. This is particularly unfortunate because, in many cases, the monitor itself must have internally used the same reasoning the engineer is trying to recover (from the witness trace) in order to reach its own conclusion.

In this paper, we set out to *formalise the reasoning* that is carried out by such monitors, in order to be able to study attributes of the monitoring process. We strive towards a formalism that is abstract enough to be used as a *justification device* for the verdicts reached, without revealing unnecessary details of the underlying monitoring algorithm: most RV approaches are either too abstract, simply yielding a verdict without explaining *why* that verdict was reached, or else too concrete, revealing the implementation internals of the monitor. Ideally, this formalism should also be comprehensive enough to express existing approaches that offer a degree of *explainability* [31, 42]; this will lead to a better understanding of how to best present justifications for the verdicts reach via runtime monitoring.

We focus on providing formalisms for the monitoring of properties expressed as LTL formulas. In particular, we seek to construct a *proof system* for LTL

[1] There are additional problems that are not covered in this article, such as the sheer size of the trace itself, that is often a byproduct of long periods of monitoring.

that is attuned to the constraints of an RV setting, and show how it can be used as a basis for monitor construction. Formalising monitor reasoning as a proof system is appealing because it allows us to back up a monitor verdict with a *proof derivation*, explaining how the verdict was reached. Proof systems are also a general formalism that complement well formal logics: they have been extensively studied as a means for embodying mechanical syntactic deductions [25, 64]. Although a number of deductive systems for LTL exist, *e.g.,* [53, 49, 23] they are generally geared towards reasoning about the full point-space of an LTL property. By contrast, we require our proof system to be *local* [62, 22], *i.e.,* focussing on checking whether a *specific point* lies within a property set, instead of interpreting a formula with respect to a set of points (which may be costly to calculate); this mirrors closely the runtime analysis carried out in RV.

RV settings pose a number of constraints on the runtime symbolic analysis carried out. In online settings, deductions are often performed on the *partial* traces generated thus far by the executing system under scrutiny. This carries a number of important consequences:

(a) Verdicts reached by a monitor should be *irrevocable* and *consistent with any extension* leading to a complete trace. Stated otherwise, although the monitor is required to reach a verdict using partial information, *i.e.,* the execution observed thus far, it is not allowed to change its verdict as further events of the execution trace are observed.

(b) Given the incomplete nature of the execution provided to a monitor, it is possible that the partial trace observed so far does not provide enough information to reach a conclusive verdict. However, in order to keep RV overheads low, it would be ideal if *inconclusive* deductions are reusable, and contribute to deductions of subsequent extensions *i.e.*, the analysis must be *incremental*.

(c) The inability to derive a *satisfaction verdict* from a partial trace does not necessarily imply the inability to derive a *violation verdict* for that partial trace (and vice-versa). Thus monitors for partial traces typically reason about trace *satisfactions*, but also trace *violations*[19, 18] so as to determine good/bad prefixes [50] and terminate monitoring as soon as either verdict is reached (keeping overheads lower still). Accordingly, our proof system deductions should reason directly on *both* satisfactions and violations.

(d) Timely detections often require a *synchronous* monitor instrumentation where monitored system execute in lock-step with the respective monitor, producing a trace event and waiting for the monitor to terminate its (incremental) analysis before executing further. In order for such an instrumentation to be safe, it is important to ensure that incremental deductions are at least *guaranteed to terminate*, which then ensures that the monitor will get back to the instrumented system so as to allow it to execute further.

Our work is not the first to consider (subsets of) the aforementioned RV constraints and requirements for the logic LTL; numerous others have, for instance, incorporated these constraints in the logic semantics [18]. However, considering these aspects at a proof-theoretic level offers a number of advantages. In particular, it allows us to keep the semantic definition of the logic *constant*. This leads to better separation of concerns, whereby the correctness specifications formulated in the logic are *agnostic* to the verification technique used; we could therefore change our verification method—from RV to, say, model checking—without affecting the meaning of the correctness specification. This separation also allows us to study which subsets of the logic is monitorable [5] in terms of a formal framework that abstracts away from certain details of the monitor implementation. Specifically, there are certain LTL properties that cannot be checked at runtime [3, 5], and our methodology lays a foundation to study such limits from a proof-theoretic perspective.

There are important concerns relating to whether our proposed methodology (based around a proof systems) can indeed provide explanations to a satisfactory number of cases. We answer these concerns indirectly. In fact, there is established work that is comparable to ours and can also be used as a basis for our methodology, *i.e.,* providing high-level explanations for RV verdicts reached for LTL formulas. We consider two of these works that are reasonably different from one another: Geilen's work [41] is based on the notion of informative prefixes [50], whereas Sen *et al.*'s work [61] is based on the notion of derivatives [45]. We argue that because of their differences (*e.g.,* they assume different syntactic subsets of the logic, they use different symbolic analyses, they sit at different levels of abstraction *etc.*), these formalisms are not easy to relate to one another. In this paper, we formally compare our proof system with these RV symbolic formalisms. Apart from enabling us to asses the generality of our approach in terms of expressiveness, we show how such a comparison yields a better understanding of the respectively symbolic techniques that we compare to, while also suggesting aspects for cross-fertilisation across the various formalisations.

Paper Structure. The rest of the paper is structured as follows. After briefly introducing the logic syntax and semantics, Section 2, we present our local proof system for partial traces in Section 3. We prove soundness for our proof system with respect to the logic semantics of Section 2 and also establish incompleteness results. Section 4 demonstrates how a monitoring algorithm can be obtained from this proof system; we show that the resulting algorithm is incremental, decidable and produces irrevocable verdicts. In Section 5 we study the expressiveness of our proof system by establishing formal comparisons with other RV symbolic analyses. Section 6 concludes with a summary of our contributions and a discussion of other work that is related to ours.

2. The Logic: An LTL Primer

As is common to most verifications setups, RV correctness properties are typically expressed as formulas from a formal logic, whose semantics describes the desired system behaviour. Apart from providing a precise meaning, the regular syntax of the properties expressed in such logics also facilitates the automation of the monitor synthesis procedure; see [1, 3] for a detailed study of this. In an RV setting, system behaviour is often described as *sets of execution traces*, because it agrees with the type of observations carried out by a monitor at runtime. In fact, Linear Temporal Logic, (LTL) [57] (and its variants) is prevalently used to specify correctness properties in formal expositions of RV [41, 32, 61, 17, 18, 19, 16, 15], due to its pleasingly straightforward semantic definition over strings denoting execution traces. We give a brief outline of this logic in terms of its syntax and semantics.

2.1. The Syntax

Figure 1 defines the *core* syntax of Linear Temporal Logic (LTL) as used in RV studies such as [19, 32], parameterised by a set of predicates $p \in \textsc{Pred}$. The grammar consists of two base cases, *i.e.,* the true formula, tt, and a predicate formula, p, standard negation and conjunction constructors, $\neg\psi$ and $\psi_1 \wedge \psi_2$, and the characteristic *next* and *until* formulas, $\mathsf{X}\psi$ and $\psi_1 \; \mathsf{U} \; \psi_2$ respectively.

Some studies using LTL (*e.g.,* [41, 23]) prefer to work with formulas in *negation normal form* (nnf), where negations are pushed to the leaves of a formula. To accommodate this, we also consider an extended LTL syntax in Figure 1, that also includes base formulas for falsehood, ff, and constructors such as disjunctions, $\varphi_1 \vee \varphi_2$, and release formulas, $\varphi_1 \; \mathsf{R} \; \varphi_2$. Our extended syntax also employs an extended predicate notation that includes *co-predicates*, *i.e.,* for any predicate p, its co-predicates, denoted as $\overline{p}$, represents another predicate that acts as its *dual* (whenever p returns true, $\overline{p}$ returns false, and vice-versa). This allows us to eliminate negations from normalised formulas and, because of this, we sometimes refer to an nnf formula as *negation-free.*[2] Figure 1 also defines a translation function, $\langle - \rangle :: \text{LTL} \rightarrow \text{eLtl}$ from formulas of the core LTL to a negation-free formula in the extended syntax.

2.2. The Model

The logic semantics is also given in Figure 1. It assumes an alphabet, Σ (with element variables σ), over which predicates are defined, $p :: \Sigma \rightarrow \textsc{Bool}$. In the rest of the paper, predicate definitions are denoted as sets over Σ, *i.e.,* $S \subseteq \Sigma$ where, accordingly, the co-predicate for a predicate $p = S$ is defined as $\Sigma \setminus S$. The truth value of a predicate p (respectively co-predicate $\overline{p}$) for element σ is denoted as $p(\sigma)$ (respectively $\overline{p}(\sigma)$).

As in most RV studies, such as the work in [41, 61, 19, 44], the logic is defined over *infinite* strings, $s \in \Sigma^\omega$, denoting execution traces. *Finite* strings over the same alphabet represent *partial* executions and are denoted by the variable $t \in \Sigma^*$; the symbol ϵ is used to represent the empty string. A string with element σ at its head is denoted as σs (and respectively σt for finite strings). For indexes $i, j \in \textsc{Nat}$, s_i denotes the i^{th} *element* in the string (starting from index 0) and $[s]^i$ denotes the *suffix* of s starting at index i; for finite strings

[2] Co-predicates have also been used in other expositions of LTL such as [23, 5].

Core LTL Syntax

$$
\begin{array}{rcllllll}
\psi \in \mathrm{LTL} & ::= & \mathsf{tt} & \textit{(true)} & \mid\ p & \textit{(predicate)} & \mid\ \psi_1 \wedge \psi_2 & \textit{(conjuction)} \\
& \mid & \neg\psi & \textit{(negation)} & \mid\ \psi_1 \ \mathsf{U}\ \psi_2 & \textit{(until)} & \mid\ \mathsf{X}\psi & \textit{(next)}
\end{array}
$$

Extended LTL Syntax

$$
\begin{array}{rcllllll}
\varphi \in \mathrm{E}\textsc{Ltl} & ::= & \mathsf{tt} & & \mid\ p & & \mid\ \varphi_1 \wedge \varphi_2 & \\
& \mid & \varphi_1 \ \mathsf{U}\ \varphi_2 & & \mid\ p\mathsf{X}\varphi & & \mid\ \neg\varphi & \\
& \mid & \mathsf{ff} & \textit{(false)} & \mid\ \overline{p} & \textit{(co-predicate)} & \mid\ \varphi_1 \vee \varphi_2 & \textit{(disjunction)} \\
& \mid & \varphi_1 \ \mathsf{R}\ \varphi_2 & \textit{(release)} & & & &
\end{array}
$$

Formula Translation (Normalisation)

$$
\begin{array}{rlrl}
\langle \mathsf{tt} \rangle & \stackrel{\text{def}}{=} \mathsf{tt} & \langle \neg\mathsf{tt} \rangle & \stackrel{\text{def}}{=} \mathsf{ff} \\
\langle p \rangle & \stackrel{\text{def}}{=} p & \langle \neg p \rangle & \stackrel{\text{def}}{=} \overline{p} \\
\langle \mathsf{X}\psi \rangle & \stackrel{\text{def}}{=} \mathsf{X}\langle \psi \rangle & \langle \neg\mathsf{X}\psi \rangle & \stackrel{\text{def}}{=} \mathsf{X}\langle \neg\psi \rangle \\
\langle \neg\neg\psi \rangle & \stackrel{\text{def}}{=} \langle \psi \rangle & \langle \psi_1 \wedge \psi_2 \rangle & \stackrel{\text{def}}{=} \langle \psi_1 \rangle \wedge \langle \psi_2 \rangle \\
\langle \psi_1 \ \mathsf{U}\ \psi_2 \rangle & \stackrel{\text{def}}{=} \langle \psi_1 \rangle \ \mathsf{U}\ \langle \psi_2 \rangle & \langle \neg\big(\psi_1 \ \mathsf{U}\ \psi_2\big) \rangle & \stackrel{\text{def}}{=} \langle \neg\psi_1 \rangle \ \mathsf{R}\ \langle \neg\psi_2 \rangle \\
\langle \neg\big(\psi_1 \wedge \psi_2\big) \rangle & \stackrel{\text{def}}{=} \langle \neg\psi_1 \rangle \vee \langle \neg\psi_2 \rangle & &
\end{array}
$$

Semantics

$$
\begin{array}{rlrl}
[\![\mathsf{tt}]\!] & \stackrel{\text{def}}{=} \Sigma^\omega & [\![\mathsf{ff}]\!] & \stackrel{\text{def}}{=} \emptyset \\
[\![p]\!] & \stackrel{\text{def}}{=} \{s \mid p(s_0)\} & [\![\overline{p}]\!] & \stackrel{\text{def}}{=} \{s \mid \text{not } p(s_0)\} \\
[\![\varphi_1 \wedge \varphi_2]\!] & \stackrel{\text{def}}{=} [\![\varphi_1]\!] \cap [\![\varphi_1]\!] & [\![\varphi_1 \vee \varphi_2]\!] & \stackrel{\text{def}}{=} [\![\varphi_1]\!] \cup [\![\varphi_1]\!] \\
[\![\mathsf{X}\psi]\!] & \stackrel{\text{def}}{=} \big\{s \mid [s]^1 \in [\![\psi]\!]\big\} & [\![\neg\varphi]\!] & \stackrel{\text{def}}{=} \big(\Sigma^\omega\big) \setminus [\![\varphi]\!] \\
[\![\varphi_1 \ \mathsf{U}\ \varphi_2]\!] & \multicolumn{3}{l}{\stackrel{\text{def}}{=} \big\{s \mid \exists j \text{ such that } [s]^j \in [\![\varphi_2]\!] \text{ and } (\forall i. i < j \text{ implies } [s]^i \in [\![\varphi_1]\!])\big\}} \\
[\![\varphi_1 \ \mathsf{R}\ \varphi_2]\!] & \multicolumn{3}{l}{\stackrel{\text{def}}{=} \big\{s \mid \forall j \text{ we have } \big([s]^j \in [\![\varphi_2]\!] \text{ or } (\exists i. i < j \text{ such that } [s]^i \in [\![\varphi_1]\!])\big)\big\}}
\end{array}
$$

Figure 1: Linear Temporal Logic Syntax and Semantics

t we have the (implicit) condition that the suffix index satisfies $i \leq |t|$, since $[t]^{|t|} = \epsilon$. Note that for any s, the suffix at index 0 acts as the identity function, *i.e.*, $[s]^0 = s$. Infinite strings with a regular (finite) pattern t are sometimes denoted as t^ω, whereas the shorthand $t\ldots$ represents some infinite string with a (finite) prefix t.

2.3. The Semantics

The denotational semantic function $[\![-]\!] :: \text{eLTL} \rightarrow \mathcal{P}(\Sigma^\omega)$ is defined by induction over the structure of LTL formulas; in Figure 1 we define the semantics for the extended LTL syntax (of which the core syntax is a subset). Most cases are standard. For instance, $[\![\mathsf{tt}]\!]$ (respectively $[\![\mathsf{ff}]\!]$) returns the universal (respectively empty) set of (infinite) strings defined over the alphabet Σ, $[\![\neg\varphi]\!]$ returns the dual of $[\![\varphi]\!]$, whereas $[\![\varphi_1 \wedge \varphi_2]\!]$ (respectively $[\![\varphi_1 \vee \varphi_2]\!]$) denotes the intersection (respectively union) of the meaning of its sub-formulas, $[\![\varphi_1]\!]$ and $[\![\varphi_2]\!]$. The meaning of $[\![p]\!]$ contains all strings s whose *first element* s_0 satisfies the predicate p *i.e.*, $p(s_0)$ returns true; dually, $[\![\overline{p}]\!]$ contains all strings whose first elements violates p, *i.e.*, $p(s_0)$ returns false. The temporal formulas are more involving. The denotation of $\mathsf{X}\varphi$ contains all strings whose immediate suffix (*i.e.*, at index 1) is included in $[\![\psi]\!]$. Until formulas $[\![\varphi_1 \;\mathsf{U}\; \varphi_2]\!]$ contain all strings that contain a suffix (at *some* index j) satisfying $[\![\psi_2]\!]$, and *all* the suffixes preceding j satisfy $[\![\psi_1]\!]$. Finally, release formulas, $\varphi_1 \;\mathsf{R}\; \varphi_2$ contain strings whose suffixes *always* satisfy φ_2, as well as strings that contain a suffix satisfying both φ_1 and φ_2 and *all* the preceding suffixes satisfying φ_2.

The denotational semantics allows us to observe the duality between the formulas tt, $\varphi_1 \wedge \varphi_2$ and $\psi_1 \;\mathsf{U}\; \psi_2$, and their counterparts ff, $\varphi_1 \vee \varphi_2$ and $\psi_1 \;\mathsf{R}\; \psi_2$. It also helps us understand the mechanics of the translation function, pushing negation to the leaves of a formula using negation propagation identities (*e.g.*, DeMorgan's law), converting constructors to their dual constructor; at the leaves the function then performs direct translations from tt and p to ff and $\overline{p}$ respectively. The semantics also allows us to prove Proposition 1, justifying the use of a corresponding negation-free formula (with the same meaning) instead of a core LTL formula, in order to reason exclusively in terms of positive interpretations. Proposition 1 (implicitly) states that (i) the translation function is *total* (otherwise the equality cannot be determined) but also states that (ii) the translated formula *preserves the semantic meaning* of the original formula.

Proposition 1. *For any* $\psi \in \text{LTL}$, $[\![\psi]\!] = [\![\langle\psi\rangle]\!]$

Proof. We prove an alternative statement from which the required result follows; we show that

$$\text{for any } \psi \in \text{LTL},\ [\![\psi]\!] = [\![\langle\psi\rangle]\!] \text{ and } [\![\neg\psi]\!] = [\![\langle\neg\psi\rangle]\!].$$

The proof proceeds by induction on the structure of ψ. We here consider the sub-case where $\psi = \psi_1 \;\mathsf{U}\; \psi_2$ and present the proof for the second clause, *i.e.*, we aim to show that $[\![\neg\psi]\!] = [\![\langle\neg\psi\rangle]\!]$. From Figure 1, by expanding $[\![\neg(\psi_1 \mathsf{U} \psi_2)]\!]$ we obtain

$$\begin{aligned}\left(\Sigma^\omega\right) \setminus \left\{s \mid \exists j \text{ such that } [s]^j \in [\![\psi_2]\!] \text{ and } (i < j \text{ implies } [s]^i \in [\![\psi_1]\!])\right\} \\ = \quad \left\{s \mid \forall j \text{ we have } \left([s]^j \notin [\![\psi_2]\!] \text{ or } (\exists i < j \text{ such that } [s]^i \notin [\![\psi_1]\!])\right)\right\}\end{aligned} \tag{1}$$

From the definition of the translation function in Figure 1, we know that $\langle\neg(\psi_1 \mathsf{U} \psi_2)\rangle = \langle\neg\psi_1\rangle \mathsf{R} \langle\neg\psi_2\rangle$. Moreover, by expanding $[\![\langle\neg\psi_1\rangle \mathsf{R} \langle\neg\psi_2\rangle]\!]$ we obtain.

$$\left\{s \mid \forall j \text{ we have } \left([s]^j \in [\![\langle\neg\psi_2\rangle]\!] \text{ or } (\exists i < j \text{ such that } [s]^i \in [\![\langle\neg\psi_1\rangle]\!])\right)\right\} \tag{2}$$

Now we know $[s]^j \notin [\![\psi_2]\!]$ iff $[s]^j \in [\![\neg\psi_2]\!]$, and similarly $[s]^i \notin [\![\psi_1]\!]$ iff $[s]^i \in [\![\neg\psi_1]\!]$. Moreover, by I.H. we can obtain that $[\![\neg\psi_1]\!] = [\![\langle\neg\psi_1\rangle]\!]$ and $[\![\neg\psi_2]\!] = [\![\langle\neg\psi_2\rangle]\!]$, from which we can equate the two sets (1) and (2). □

Example 2.1. The behaviour of a traffic-light system may be described by regularly observing its states as a set of traces consisting of green, g, orange, o, and red, r. Complete executions may thus be represented as traces (infinite strings) over the alphabet $\Sigma = \{g, o, r\}$. Predicate definitions may be described as sets over this alphabet Σ, *e.g.,* $\mathsf{st} = \{o, r\}$ is true only for then *stopping* actions o and r whereas $\mathsf{mv} = \{o, g\}$ represent states where vehicles may be *in movement*; singleton-set predicates are denoted by the single-letter names convention *e.g.,* $\mathsf{g} = \{g\}$. Using the logic of Figure 1, we can specify the following properties:

- $(\neg\mathsf{r}) \land \mathsf{X}\mathsf{r}$ describes a trace where the system is not in a red state initially, but turns red at the next instant. Traces of the form $gr\ldots$ and $or\ldots$ satisfy this property, but others such as $ggg\ldots$ or $ro\ldots$ do not. Note also that whereas the trace $ggg\ldots$ violates the first subformula, $\neg\mathsf{r}$, the second trace $ro\ldots$ violates $\mathsf{X}\mathsf{r}$. This information can be crucial for debugging;

- $\mathsf{g} \;\mathsf{U}\; \mathsf{o}$ describes traces that eventually switch to the orange state from a green state. Traces of the form $gggo\ldots$ satisfy this property, whereas other such as $ggr\ldots$ or g^ω do not;

- $\mathsf{st} \;\mathsf{U}\; \neg\mathsf{st}$ describes traces that reach a *non-stopping* action after a sequence of *stopping* actions. Using the respective semantic definitions, one can check that $[\![\mathsf{st} \;\mathsf{U}\; \mathsf{g}]\!] \subseteq\supseteq [\![\mathsf{st} \;\mathsf{U}\; \neg\mathsf{st}]\!] \subseteq\supseteq [\![\big((\mathsf{mv} \land \mathsf{o}) \lor \mathsf{r}\big) \;\mathsf{U}\; \neg\mathsf{st}]\!]$ ($\subseteq\supseteq [\![\mathsf{F}\,\neg\mathsf{st}]\!]$ where F denotes the *"eventually"* operator). Occasionally, it might be easier to check for one description of the property as opposed to the others.

- $\mathsf{G}\,\mathsf{st}$, *i.e.,* always st, which is shorthand for $\neg(\mathsf{tt} \;\mathsf{U}\; \neg\mathsf{st})$ [10], describes traces that contain only stopping states. Traces of the form $(or)^\omega$ and r^ω are included in the property whereas a trace such as $oroog\ldots$ is not. Again, for debugging purposes, it is may be useful to know that the property was violated at the fifth iteration of the invariance check since $g \notin \mathsf{st}$.

The semantics provides multiple ways of checking for a formula. To determine whether the complete trace r^ω satisfies formula $\mathsf{G}\,\mathsf{st}$, *i.e.*, $r^* \in [\![\mathsf{G}\,\mathsf{st}]\!]$, we can use the semantic definition of the formula to compute the set $[\![\neg(\mathsf{tt}\ \mathsf{U}\ \neg\mathsf{st})]\!]$ for which we would need to first calculate $[\![\mathsf{tt}\ \mathsf{U}\ \neg\mathsf{st}]\!]$ and then take its dual. Alternatively, we can calculate the denotation of $\langle\neg(\mathsf{tt}\ \mathsf{U}\ \neg\mathsf{st})\rangle$, which translates to $\mathsf{ff}\ \mathsf{R}\ \mathsf{st}$, and check inclusion with respect to the translated negation-free formula, safe in the knowledge that $[\![\neg(\mathsf{tt}\ \mathsf{U}\ \neg\mathsf{st})]\!] = [\![\mathsf{ff}\ \mathsf{R}\ \mathsf{st}]\!]$ (by Proposition 1). Using similar reasoning, to determine whether the violation $r \ldots \notin [\![(\neg\mathsf{r}) \wedge \mathsf{X}\mathsf{r}]\!]$ holds, we can instead check whether the membership $r \ldots \in [\![\mathsf{r} \vee \mathsf{X}\bar{\mathsf{r}}]\!]$ holds. ■

We note an important aspect of the semantics in Figure 1 which has repercussions on our RV perspective. In the case of $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ from Example 2.1, we require a *global view* of a trace such as g^ω in order to determine that it violates property $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$. Stated otherwise, *no finite prefix* of g^ω yields enough information to be able to conclude the violation. By contrast, there exists a finite prefix of the complete trace $ggr\ldots$ that allows us to conclude $ggr\ldots \notin [\![\mathsf{g}\ \mathsf{U}\ \mathsf{o}]\!]$, namely the prefix ggr. Moreover, all traces satisfying $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ contain a finite prefix that allows us to determine the respectively satisfaction. In the case of $\mathsf{G}\,\mathsf{st}$ from Example 2.1, the situation is almost the reverse: whereas satisfactions can only be determined through a global view of the trace, violations can always be determined by observing a finite prefix.

3. An Online Monitoring Proof System

Online[3] runtime verification of LTL properties consists in determining whether the current execution satisfies (or violates) a property from the *finite* trace generated thus far. We present a local proof system [62, 22] that characterises this runtime analysis, and allows us to determine whether any complete trace ts with *finite* prefix t is included in (or excluded from) $[\![\varphi]\!]$. The proof system is defined as the least relation satisfying the rules in Figure 2. These rules employ two, mutually dependent, judgements: the sequent $t \vdash^+ \varphi$ denotes a *satisfaction* judgement, whereas $t \vdash^- \psi$ denotes a *violation* judgement; note the polarity differentiating the two judgements, *i.e.*, the annotations $+$ and $-$.

Figure 2 includes three satisfaction axioms (pTru, pPrd and pCoP) and three violation axioms (nFls, nPrd and nCoP); the proof system is parametric with respect to the pre-computation of predicates and co-predicates, p and $\overline{p}$. The conjunction and disjunction rules, pAnd, pOr1 and pOr2 (respectively nAnd1, nAnd2 and nOr) *decompose* the composite formula of the judgement for their premises. The negation rules pNeg and nNeg also decompose the formula, but *switch the modality* of the sequents for their premises, *transitioning* from one judgement form to the other. Specifically, in the case of pNeg, the satisfaction sequent $t \vdash^+ \neg\varphi$ is defined in terms of the *violation* sequent $t \vdash^- \varphi$ (and dually for nNeg).

[3] By contrast, *offline* monitoring typically works on *complete* execution traces [59].

Satisfaction Rules

$$\textsc{pTru}\,\frac{}{t \vdash^{+} \mathsf{tt}} \qquad \textsc{pNeg}\,\frac{t \vdash^{-} \varphi}{t \vdash^{+} \neg\varphi}$$

$$\textsc{pPrd}\,\frac{p(\sigma)}{\sigma t \vdash^{+} p} \qquad \textsc{pCoP}\,\frac{\overline{p}(\sigma)}{\sigma t \vdash^{+} \overline{p}}$$

$$\textsc{pAnd}\,\frac{t \vdash^{+} \varphi_1 \quad t \vdash^{+} \varphi_2}{t \vdash^{+} \varphi_1 \wedge \varphi_2} \qquad \textsc{pNxt}\,\frac{t \vdash^{+} \varphi}{\sigma t \vdash^{+} \mathsf{X}\varphi}$$

$$\textsc{pOr1}\,\frac{t \vdash^{+} \varphi_1}{t \vdash^{+} \varphi_1 \vee \varphi_2} \qquad \textsc{pOr2}\,\frac{t \vdash^{+} \varphi_2}{t \vdash^{+} \varphi_1 \vee \varphi_2}$$

$$\textsc{pUnt1}\,\frac{t \vdash^{+} \varphi_2}{t \vdash^{+} \varphi_1 \;\mathsf{U}\; \varphi_2} \qquad \textsc{pUnt2}\,\frac{\sigma t \vdash^{+} \varphi_1 \quad t \vdash^{+} \varphi_1 \;\mathsf{U}\; \varphi_2}{\sigma t \vdash^{+} \varphi_1 \;\mathsf{U}\; \varphi_2}$$

$$\textsc{pRel1}\,\frac{t \vdash^{+} \varphi_1 \quad t \vdash^{+} \varphi_2}{t \vdash^{+} \varphi_1 \;\mathsf{R}\; \varphi_2} \qquad \textsc{pRel2}\,\frac{\sigma t \vdash^{+} \varphi_2 \quad t \vdash^{+} \varphi_1 \;\mathsf{R}\; \varphi_2}{\sigma t \vdash^{+} \varphi_1 \;\mathsf{R}\; \varphi_2}$$

Violation Rules

$$\textsc{nFls}\,\frac{}{t \vdash^{-} \mathsf{ff}} \qquad \textsc{nNeg}\,\frac{t \vdash^{+} \varphi}{t \vdash^{-} \neg\varphi}$$

$$\textsc{nPrd}\,\frac{\overline{p}(\sigma)}{\sigma t \vdash^{-} p} \qquad \textsc{nCoP}\,\frac{p(\sigma)}{\sigma t \vdash^{-} \overline{p}}$$

$$\textsc{nOr}\,\frac{t \vdash^{-} \varphi_1 \quad t \vdash^{-} \varphi_2}{t \vdash^{-} \varphi_1 \vee \varphi_2} \qquad \textsc{nNxt}\,\frac{t \vdash^{-} \psi}{\sigma t \vdash^{-} \mathsf{X}\psi}$$

$$\textsc{nAnd1}\,\frac{t \vdash^{-} \varphi_1}{t \vdash^{-} \varphi_1 \wedge \varphi_2} \qquad \textsc{nAnd2}\,\frac{t \vdash^{-} \varphi_2}{t \vdash^{-} \varphi_1 \wedge \varphi_2}$$

$$\textsc{nUnt1}\,\frac{t \vdash^{-} \varphi_1 \quad t \vdash^{-} \varphi_2}{t \vdash^{-} \varphi_1 \;\mathsf{U}\; \varphi_2} \qquad \textsc{nUnt2}\,\frac{\sigma t \vdash^{-} \varphi_2 \quad t \vdash^{-} \varphi_1 \;\mathsf{U}\; \varphi_2}{\sigma t \vdash^{-} \varphi_1 \;\mathsf{U}\; \varphi_2}$$

$$\textsc{nRel1}\,\frac{t \vdash^{-} \varphi_2}{t \vdash^{-} \varphi_1 \;\mathsf{R}\; \varphi_2} \qquad \textsc{nRel2}\,\frac{\sigma t \vdash^{-} \varphi_1 \quad t \vdash^{-} \varphi_1 \;\mathsf{R}\; \varphi_2}{\sigma t \vdash^{-} \varphi_1 \;\mathsf{R}\; \varphi_2}$$

Figure 2: Satisfaction and Violation Proof Rules

The rules for the temporal formulas may decompose judgement formulas, *e.g.*, $\textsc{pUnt1}$, $\textsc{pRel1}$, $\textsc{nUnt1}$, $\textsc{nRel1}$, but may also analyse suffixes of the trace *in incremental fashion.* For instance, in order to prove $\sigma t \vdash^+ \mathsf{X}\varphi$, rule $\textsc{pNxt}$ requires the satisfaction judgement to hold for the *immediate* suffix t and the sub-formula φ, *i.e.,* $t \vdash^+ \varphi$. Similarly, to prove the satisfaction sequent $\sigma t \vdash^+ \varphi_1 \mathbin{\mathsf{U}} \varphi_2$, rule $\textsc{pUnt2}$ requires a satisfaction proof of the current trace σt and the sub-formula φ_1, as well as a satisfaction proof of the immediate suffix t with respect to $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$. Since this suffix premise is with respect to the same composite formula $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$, it may well be the case that $\textsc{pUnt2}$ is applied again for suffix t. In fact, satisfaction proofs for until formulas are characterised by a series of $\textsc{pUnt2}$ applications, followed by an application of rule $\textsc{pUnt1}$ (the satisfaction proofs for $\varphi_1 \mathbin{\mathsf{R}} \varphi_2$ and violation proofs for $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$ and $\varphi_1 \mathbin{\mathsf{R}} \varphi_2$ follow an analogous structure). This incremental analysis structure mirrors that of RV algorithms for LTL [41, 61, 19] and contrasts with the descriptive nature of the semantic definition for $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$ (Figure 1) (which merely stipulates the *existence* of some index j at which point φ_2 holds without stating *how* to find this index).

We note the inherent symmetry between the satisfaction and violation rules, internalising the negation-propagation mechanism of the normalisation function $\langle - \rangle$ of Section 2 through rules $\textsc{pNeg}$ and $\textsc{nNeg}$. For instance, there are no satisfaction proof rules for the formula ff and there are no violation rules for the formula tt either. The respective predicate axioms for satisfactions and violations are dual to one another, as are the rules for conjunctions and disjunctions. More precisely, following $\langle \neg(\psi_1 \wedge \psi_2) \rangle \stackrel{\text{def}}{=} \langle \neg\psi_1 \rangle \vee \langle \neg\psi_2 \rangle$ from Figure 1, the violation rules for conjunctions ($\textsc{nAnd1}$ and $\textsc{nAnd2}$) have the same structure as the satisfaction rules for the respective disjunctions ($\textsc{pOr1}$ and $\textsc{pOr2}$). The symmetric structure carries over to the temporal proof rules as well, *e.g.*, violation rules for the until formulas, $\textsc{nUnt1}$ and $\textsc{nUnt2}$, have an analogous structure to that of the release formula rules, $\textsc{pRel1}$ and $\textsc{pRel2}$.

Example 3.1. Recall property $\mathsf{g} \mathbin{\mathsf{U}} \mathsf{o}$ from Ex. 2.1. We can construct the satisfaction proof for trace go and the violation proof for trace gr below:

$$\textsc{pUnt2}\ \dfrac{\textsc{pPrd}\ \dfrac{\mathsf{g}(g)}{go \vdash^+ \mathsf{g}} \qquad \textsc{pUnt1}\ \dfrac{\textsc{pPrd}\ \dfrac{\mathsf{o}(o)}{o \vdash^+ \mathsf{o}}}{o \vdash^+ \mathsf{g} \mathbin{\mathsf{U}} \mathsf{o}}}{go \vdash^+ \mathsf{g} \mathbin{\mathsf{U}} \mathsf{o}} \qquad \textsc{nUnt2}\ \dfrac{\textsc{nPrd}\ \dfrac{\bar{\mathsf{o}}(g)}{gr \vdash^- \mathsf{o}} \qquad \textsc{nUnt1}\ \dfrac{\textsc{nPrd}\ \dfrac{\bar{\mathsf{g}}(r)}{r \vdash^- \mathsf{g}} \qquad \textsc{nPrd}\ \dfrac{\bar{\mathsf{o}}(r)}{r \vdash^- \mathsf{o}}}{r \vdash^- \mathsf{g} \mathbin{\mathsf{U}} \mathsf{o}}}{gr \vdash^- \mathsf{g} \mathbin{\mathsf{U}} \mathsf{o}}$$

Crucially, however, we are *unable* to construct *any* proof for the finite trace gg. For instance, attempting to construct a satisfaction proof fails because we hit the end-of-trace, ϵ, before completing the proof tree. Intuitively, we do not have enough information from the trace generated thus far to conclude that the complete trace satisfies the property. For instance, the next state may be o, in which case we can infer satisfaction for ggo, or it can be r, in which case we infer a violation for ggr; the next state may also be g, in which case we have to postpone any conclusive judgement once again.

$$
\textsc{pUnt2}\;\dfrac{\textsc{pPrd}\;\dfrac{\mathsf{g}(g)}{gg \vdash^{+} \mathsf{g}} \qquad \textsc{pUnt2}\;\dfrac{\textsc{pPrd}\;\dfrac{\mathsf{g}(g)}{g \vdash^{+} \mathsf{g}} \qquad ??\;\dfrac{}{\epsilon \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}}}{g \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}}}{gg \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}}
$$

We also note that although we can construct a satisfaction proof for the judgement $go \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$, we are unable to construct a violation proof for the same partial trace and formula. The failed derivation below shows how we necessarily get stuck trying to prove the sequent $o \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$. The reason for this is because, due to the structure of the formula and the sequent polarity, we are forced to use either rule nUnt1 or rule nUnt2; both of these rules require us to prove $o \vdash^{-} \mathsf{o}$ as one of their premises, which we clearly cannot do because the only rule that we can apply, nPrd, requires $\bar{\mathsf{o}}(o)$, which clearly does *not* hold.

$$
\textsc{nUnt2}\;\dfrac{\textsc{nPrd}\;\dfrac{\bar{\mathsf{o}}(g)}{go \vdash^{-} \mathsf{o}} \qquad ??\;\dfrac{}{o \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}}}{go \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}}
$$

In fact, it turns out that we can never both derive a satisfaction proof *and* a violation proof for *any* partial trace and formula pair (*e.g.,* in the example above, we could not derive proofs for both judgements $go \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ and $go \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$). This is an important sanity check that ensures the *consistency* of our derivations.

We also point out another important aspect from the example derivations above. In particular, there is a difference between the failed derivation proof for the judgement $gg \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ and the failed derivation for the judgement $go \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$. In both cases, the derivations reached a point where no rules could be applied to a derivation branch. However, the reason why no rule could be applied is different: whereas the derivation for $gg \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ could not be completed because the (partial) trace was *not long enough i.e.,* we hit the end of string ϵ, the derivation for $go \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ failed *before* we reached the end of the (partial) trace *i.e.,* at stage o. Thus, in a setting where we may learn more parts of the trace in future, we should keep the incomplete derivation for judgement $gg \vdash^{+} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ as this may be extended to a completed derivation. By contrast, there is no extension to the trace go that may yield a completed derivation for the judgement $go \vdash^{-} \mathsf{g}\ \mathsf{U}\ \mathsf{o}$. We revisit this point later in Section 4. ■

3.1. Properties of the Proof System

We recall that the semantics of our LTL formulas was defined over infinite strings (complete traces), whereas our proof system associates *finite* traces to LTL formulas. In spite of this apparent mismatch, we can show that our proof system is sound, in the following sense:

- whenever we can construct a satisfaction proof for a prefix t and an LTL formula φ, then we know that for *any* (infinite) extension s of the prefix, the resulting (complete) trace, ts, satisfies the formula, *i.e.,* $ts \in [\![\varphi]\!]$.

- dually, whenever we can construct a violation proof for a prefix t and an LTL formula φ, then we know that for *any* extension s of the prefix, the resulting trace, ts, violates the formula,*i.e.,* $ts \notin [\![\varphi]\!]$.

Theorem 1 (Soundness). *For arbitrary t, φ:*

- $t \vdash^{+} \varphi$ *implies* $\forall s.\, ts \in [\![\varphi]\!]$
- $t \vdash^{-} \varphi$ *implies* $\forall s.\, ts \notin [\![\varphi]\!]$

Proof. By rule induction on both $t \vdash^{+} \varphi$ and $t \vdash^{-} \varphi$. Given that both of the sequents are mutually dependent on one another, we need to prove both statements simultaneously. We here show a few of the main cases.

pNeg: From the rule we know that $\varphi = \neg\varphi_1$, and that the conclusion judgement has the form $t \vdash^{+} \neg\varphi_1$. From the rule premise, $t \vdash^{-} \varphi_1$ and by I.H. we know that $\forall s.ts \notin [\![\varphi_1]\!]$. By the definition of $[\![-]\!]$ in Figure 1, this implies that $\forall s.ts \in [\![\neg\varphi_1]\!]$ as required.

pUnt2: From the rule we know $\varphi = \varphi_1 \mathsf{U} \varphi_2$. From the rule premises $\sigma t \vdash^{+} \varphi_1$ and $t \vdash^{+} \varphi_1 \mathsf{U} \varphi_2$, and by I.H. we know

$$\forall s.\, \sigma ts \in [\![\varphi_1]\!] \quad \text{and} \tag{3}$$

$$\forall s.\, ts \in [\![\varphi_1 \mathsf{U} \varphi_2]\!] \tag{4}$$

By (4) and the semantic definition of Figure 1 we obtain

$$\forall s\big(\exists\, l \geq 0\ [ts]^l \in [\![\varphi_2]\!]\ \wedge\ (j < l\ \text{ implies }\ [ts]^j \in [\![\varphi_1]\!])\big) \tag{5}$$

Thus, from (3) and (5) we know that that there exists an index, namely $k = l + 1$, such that

$$\forall s\big(\exists\, k \geq 0\ [\sigma ts]^k \in [\![\varphi_2]\!]\ \wedge\ (j \leq l\ \text{ implies }\ [\sigma ts]^j \in [\![\varphi_1]\!])\big) \tag{6}$$

which implies $\forall s.\, \sigma ts \in [\![\varphi_1 \mathsf{U} \varphi_2]\!]$ as required.

pRel2: From the rule we know $\varphi = \varphi_1 \mathsf{R} \varphi_2$. From the rule premises $\sigma t \vdash^{+} \varphi_2$ and $t \vdash^{+} \varphi_1 \mathsf{R} \varphi_2$, and by I.H. we know

$$\forall s.\, \sigma ts \in [\![\varphi_2]\!] \quad \text{and} \tag{7}$$

$$\forall s.\, ts \in [\![\varphi_1 \mathsf{R} \varphi_2]\!] \tag{8}$$

By (8) and the semantic definition of Figure 1 we obtain

$$\forall s\ \big(\forall j \text{ we have } \big([ts]^j \in [\![\varphi_2]\!] \text{ or } (\exists i < j \text{ such that } [ts]^i \in [\![\varphi_1]\!])\big)\big) \tag{9}$$

Thus, we know that either one of the following two statements, should hold true

$$\forall s.\ \forall j [ts]^j \in [\![\varphi_2]\!] \tag{10}$$

$$\forall s.\ \forall j\ \exists\ i < j \text{ such that } [ts]^i \in [\![\varphi_1]\!] \tag{11}$$

In case of (10), it immediately follows that the statement holds. In case of (11) in conjunction with (7), we know that that there exists an index, namely $k = j + 1$, such that

$$\forall s\ (\forall j \text{ we have } ([\sigma ts]^k \in [\![\varphi_2]\!] \text{ or } (\exists i < k \text{ such that } [\sigma ts]^k \in [\![\varphi_1]\!]))) \tag{12}$$

which implies $\forall s.\, \sigma ts \in [\![\varphi_1 \mathsf{R} \varphi_2]\!]$ as required.

nRel2: From the rule we know $\varphi = \varphi_1 \mathsf{R} \varphi_2$. From the rule premises $\sigma t \vdash^- \varphi_1$ and $t \vdash^- \varphi_1 \mathsf{R} \varphi_2$, and by I.H. we know

$$\forall s.\, \sigma t s \notin [\![\varphi_1]\!] \quad \text{and} \tag{13}$$

$$\forall s.\, ts \notin [\![\varphi_1 \mathsf{R} \varphi_2]\!] \tag{14}$$

By (14) and the semantic definition of Figure 1 we obtain

$$\forall s \;\big(\forall j \text{ we have } \big([ts]^j \notin [\![\varphi_2]\!] \text{ or } (\exists i < j \text{ such that } [ts]^i \notin [\![\varphi_1]\!])\big)\big) \tag{15}$$

Thus, from (13) and (15) we know that that there exists an index, namely $k = j + 1$, such that

$$\forall s \;(\forall j \text{ we have } ([\sigma t s]^k \notin [\![\varphi_2]\!] \text{ or } (\exists i < k \text{ such that } [\sigma t s]^k \notin [\![\varphi_1]\!]))) \tag{16}$$

which implies $\forall s.\, \sigma t s \notin [\![\varphi_1 \mathsf{R} \varphi_2]\!]$ as required. □

Example 3.2. As a result of Theorem 1, the satisfaction and violation proofs of Example 3.1 suffice to prove $gos \in [\![\mathsf{g} \;\mathsf{U}\; \mathsf{o}]\!]$ and $grs \notin [\![\mathsf{g} \;\mathsf{U}\; \mathsf{o}]\!]$ for any (infinite) suffix s. Moreover, to determine whether $r \ldots \notin [\![(\neg \mathsf{r}) \wedge \mathsf{X}\mathsf{r}]\!]$ from Example 2.1, it suffices to consider the prefix r and either construct a violation proof directly, or else normalise the *negation* of the formula, $\langle \neg((\neg\mathsf{r}) \wedge \mathsf{X}\mathsf{r}) \rangle = \mathsf{r} \vee \mathsf{X}\bar{\mathsf{r}}$ and construct a *satisfaction* proof:

$$\dfrac{\dfrac{\dfrac{\mathsf{r}(r)}{r \vdash^+ \mathsf{r}}\,\textsc{pPre}}{r \vdash^- \neg\mathsf{r}}\,\textsc{pNeg}}{r \vdash^- (\neg\mathsf{r}) \wedge \mathsf{X}\mathsf{r}}\,\textsc{nAnd1} \qquad\qquad \dfrac{\dfrac{\mathsf{r}(r)}{r \vdash^+ \mathsf{r}}\,\textsc{pPre}}{r \vdash^+ \mathsf{r} \vee \mathsf{X}\bar{\mathsf{r}}}\,\textsc{pOr1}$$

Note how the derivations explain which subformula was violated in the left proof, *i.e.*, $\neg\mathsf{r}$, and satisfied in the right proof, *i.e.*, r. ■

Remark 3.1. The apparent redundancy (see the two derivations in Example 3.2 for showing that $r \ldots \notin [\![(\neg\mathsf{r}) \wedge \mathsf{X}\mathsf{r}]\!]$) gives us the flexibility to use the proof system as a unifying framework that embeds other approaches (*cf.* Section 5), which may either handle negation directly in a core LTL subset, as in the case of [61], or else work exclusively with formulas in nnf, as in the case of [41]. ■

Our proof system handles empty strings ϵ, as these arise naturally from the incremental analysis of finite traces discussed above. For instance, there are cases where ϵ is required to complete a derivation, as shown below in Example 3.3. In the case of failed derivations, ϵ also plays an important role in differentiating between failed derivations, as discussed already in Example 3.1.

Example 3.3. We can prove $oo \ldots \in [\![\mathsf{X}\,\mathsf{X}\,\mathsf{tt}]\!]$ from the prefix oo, by constructing the proof tree below; the leaf node relies on being able to deduce $\epsilon \vdash^+ \mathsf{tt}$:

$$\dfrac{\dfrac{\dfrac{}{\epsilon \vdash^+ \mathsf{tt}}\,\textsc{pTru}}{o \vdash^+ \mathsf{X}\,\mathsf{tt}}\,\textsc{pNxt}}{oo \vdash^+ \mathsf{X}\,\mathsf{X}\,\mathsf{tt}}\,\textsc{pNxt}$$

■

The proof system is incomplete in the sense of Theorem 2. For instance, any (complete) trace satisfies formula $\mathsf{X}\,\mathsf{X}\,\mathsf{tt}$ (from Example 3.3) but we require at least a prefix of length 2 to determine this in our proof system.

Theorem 2 (Incompleteness)**.** *For arbitrary t, φ:*

- $\forall s.\, ts \in [\![\varphi]\!]$ *does not imply* $t \vdash^{+} \varphi$.
- $\forall s.\, ts \notin [\![\varphi]\!]$ *does not imply* $t \vdash^{-} \varphi$

Proof. By counter example. For the positive case (*i.e.*, satisfaction proofs), consider $t = \epsilon$. We can then show the following counter examples:

- $\forall s.\, ts \in [\![\mathsf{X}\,\mathsf{tt}]\!]$ but $t \nvdash^{+} \mathsf{X}\,\mathsf{tt}$;
- $\forall s.\, ts \in [\![p \vee \bar{p}]\!]$ but $t \nvdash^{+} p \vee \bar{p}$;
- $\forall s.\, ts \in [\![\mathsf{ff}\ \mathsf{R}\ \mathsf{tt}]\!]$ but $t \nvdash^{+} \mathsf{ff}\ \mathsf{R}\ \mathsf{tt}$.

More concretely, when $t = \epsilon$ we have $ts = s \in [\![\mathsf{X}\,\mathsf{tt}]\!]$ for any s. However, we *cannot* show $\epsilon \vdash^{+} \mathsf{X}\,\mathsf{tt}$: a close inspection of the rules in Figure 2 quickly reveals that no rule can be applied: the only satisfaction rule that can be applied for a formula of the form $\mathsf{X}\,\mathsf{tt}$ is pNxt, but this requires t to be of the form $\sigma t'$. Curiously, whenever a predicate p observes the property that $p(\sigma)$ holds for *all* $\sigma \in \Sigma$, we also have

- $\forall s.\, ts \in [\![p]\!]$ but $t \nvdash^{+} p$.

Specifically, the proof system treats such a predicate p differently from tt: whereas, in the latter case, the proof system can anticipate satisfaction and accept immediately, in the case of a predicate p that always holds, the proof system requires evidence of the first trace event to evaluate p over it. Although this is superfluous for the standard LTLdomain of infinite traces, it plays an important role in the finite and infinite (finfinite) domain [3, 5]. Analogous examples can be drawn up for violation proofs. □

Remark 3.2. The completeness criterion discussed above is different from completeness as used for the study of monitorability in recent work [3, 5]: in the latter definitions, the quantifiers read $\forall s.\exists$ *some prefix t. etc.*. Rather, the traces mentioned in Theorem 2 are related to (finite) traces that positively or negatively determine a property as defined in [58, 5], and the ability to yield a verdict as early as possible. See [3, Section 4.2] for a detailed discussion of this. ■

4. Runtime Monitoring with a Proof System

An automated proof search using the rules in Figure 2 can be *syntax directed* by the formula (and the respective polarity). Indeed, for most formulas, there is only *one* rule that is applicable, whereas the exception cases have *at most* two applicable rules. A breadth-first proof search can thus be automated despite this potential for non-determinism, *i.e.*, not knowing which rule to apply.

Notation. In what follows, $(t, \varphi)^+$ and $(t, \varphi)^-$ denote the respectively outstanding proof obligations $t \vdash^+ \varphi$ and $t \vdash^- \varphi$. Since our algorithm works on partial traces, $\lceil \epsilon, \varphi \rceil^+$ and $\lceil \epsilon, \varphi \rceil^-$ are used to denote *saturated* proof obligations, where the string ϵ does not yield enough information to complete the proof search (*e.g.,* $\epsilon \vdash^+ \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ in Example 3.1). A *conjunction set* $\llparenthesis o_1, \ldots, o_n \rrparenthesis$ denotes a conjunction of proof obligations; metavariables o_i range over obligations of the form $(t, \varphi)^q$ or $\lceil t, \varphi \rceil^q$ for $q \in \{+, -\}$. A *disjunction set* $\{c_1, \ldots, c_n\}$, where c_i range over conjunction sets, denotes a disjunction of conjunction sets.[4] We employ a merge operation over disjunction sets, $\oplus$, defined below:

$$d \oplus d' \stackrel{\text{def}}{=} \{c \cup c' \mid c \in d, c' \in d'\}$$

The disjunction set $\{\llparenthesis\, \rrparenthesis\}$ acts as the *identity* for this operation , *i.e.,* $\{\llparenthesis\, \rrparenthesis\} \oplus d = d \oplus \{\llparenthesis\, \rrparenthesis\} = d$, whereas the disjunction set $\{\}$ *annihilates* such sets, *i.e.,* $\{\} \oplus d = d \oplus \{\} = \{\}$.

4.1. The Algorithm

A breadth-first proof search algorithm is described in Figure 3. Disjunction sets are used to encode the alternative proof derivations that may lead to a completed proof-tree (resulting from multiple proof rules that can be applied at certain stages of the search). Conjunction sets represent the outstanding obligations within each potential derivation.

Thus, a disjunction set with an empty conjunction set $\llparenthesis\, \rrparenthesis$ as one of its elements, denotes a *successful* search where we reached a stage in one of the possible derivations with no further obligations to prove. Dually, an empty disjunction set $\{\}$ represents a *failed* search, since there are no alternative derivations left to consider in order to complete a proof derivation. Both cases are used as terminating conditions in the search algorithm of Figure 3. The only other terminating condition for this search algorithm is when a disjunction set contains only *saturated* conjunction sets: these containing *only* saturated obligations of the form $\lceil \epsilon, \varphi \rceil^q$. The predicate

$$\mathbf{sat}(c) \stackrel{\text{def}}{=} o \in c \text{ implies } o = \lceil \epsilon, \varphi \rceil^q \text{ for some } \varphi, q$$

used in Figure 3 denotes this.

To verify whether the judgement $t \vdash^q \varphi$ holds, we initiate the function $\mathbf{exp}(-)$ with the disjunction set $\{\llparenthesis (t, \varphi)^q \rrparenthesis\}$, *i.e.,* a proof search with one potential alternative derivation requiring us to prove the single sequent, $t \vdash^q \varphi$. If none of the terminating conditions in Figure 3 are met, $\mathbf{exp}(-)$ expands each conjunction set using $\mathbf{expC}(-)$, and recurses. Conjunction set expansion consists in expanding *every* proof obligation using $\mathbf{expO}(-)$ and then and merging them using $\oplus$. Obligation expansion returns a disjunction set, where each conjunction set denotes the proof obligations resulting from the premises of the rules applied. It uses two auxiliary functions:

[4] For clarity, conjunction set notation, $\llparenthesis - \rrparenthesis$, differs from that of disjunction sets, $\{-\}$.

Disjunction set expansion

$$\mathbf{exp}(d) \stackrel{\text{def}}{=} \begin{cases} \{\llparenthesis\ \rrparenthesis\} & \text{if } \llparenthesis\ \rrparenthesis \in d \\ \{\} & \text{if } d = \{\} \\ d & \text{if } c \in d \text{ implies } \mathbf{sat}(c) \\ \mathbf{exp}(\bigcup_{c \in d} \mathbf{expC}(c)) & \text{otherwise} \end{cases}$$

Conjunction set expansion

$$\mathbf{expC}(c) \stackrel{\text{def}}{=} \bigoplus_{o \in c} \mathbf{expO}(o)$$

Proof obligation expansion

$$\mathbf{expO}(o) \stackrel{\text{def}}{=} \begin{cases} \{c \mid r \in \mathbf{rls}(\varphi, q), c = \mathbf{prm}(r, t, \varphi)\} & \text{if } o = (t, \varphi)^q \\ \{\llparenthesis\, \lceil \epsilon, \varphi \rceil^q \,\rrparenthesis\} & \text{if } o = \lceil \epsilon, \varphi \rceil^q \end{cases}$$

Figure 3: A breadth-first incremental search algorithm. Auxillary functions $\mathbf{sat}(-)$, $\mathbf{prm}(-)$ and $\mathbf{rls}(-)$ as discussed inline.

- $\mathbf{rls}(\varphi, q)$ returns a set of rule names r from Figure 2 that can be applied to obligations with the formula φ and polarity qualifier q (*e.g.*, $\mathbf{rls}(\varphi_1 \mathsf{U} \varphi_2, +) = \{\textsc{pUnt1}, \textsc{pUnt2}\}$ and $\mathbf{rls}(\mathsf{X}\varphi, -) = \{\textsc{nNxt}\}$).
- $\mathbf{prm}(r, t, \varphi)$ returns a conjunction set with the premises of rule r instantiated for the conclusion with the string t and the formula φ (*e.g.*, $\mathbf{prm}(\textsc{pUnt2}, go, \mathsf{g} \mathsf{U} \mathsf{o}) = \llparenthesis\,(go, \mathsf{g})^+, (o, \mathsf{g} \mathsf{U} \mathsf{o})^+ \,\rrparenthesis$ and $\mathbf{prm}(\textsc{pTru}, go, \mathsf{tt}) = \llparenthesis\ \rrparenthesis$). Importantly, it observed the following properties for the proof system rules in Figure 2:
  - (i) For cases such as $\mathbf{prm}(\textsc{pUnt2}, \epsilon, \mathsf{g} \mathsf{U} \mathsf{o})$ the function returns the saturated conjunction set $\llparenthesis\, \lceil \epsilon, \mathsf{g} \mathsf{U} \mathsf{o} \rceil^+ \,\rrparenthesis$ since the string ϵ prohibits the function from generating *all* the premises for the rule (note that in Figure 2, one premise requires the string to be of length ≥ 1 for it to be able to calculate the tail of the string).
  - (ii) The function is *undefined* when rule conditions are not satisfied (*e.g.*, $\mathbf{prm}(\textsc{pPrd}, g, \mathsf{o})$ is undefined since $\mathsf{o}(g)$ does not hold).
  - (iii) For any rule r and formula φ, if the function $\mathbf{prm}(r, t, \varphi)$ returns a constraint set for a particular string t, then it should also return the analogous constraint set for any extension of t, *i.e.*, $\mathbf{prm}(r, t, \varphi) = c$ implies that for any t', $\mathbf{prm}(r, tt', \varphi) = \mathbf{app}(c, t')$ where the append function on a constraint set is defined as follows:

$$\mathbf{app}(c, t') \stackrel{\text{def}}{=} \llparenthesis\,(tt', \varphi)^q \mid (t, \varphi)^q \in c \,\rrparenthesis \cup \llparenthesis\,(t', \varphi)^q \mid \lceil \epsilon, \varphi \rceil^q \in c \,\rrparenthesis$$

Note that the operation $\mathbf{expO}(-)$ acts as a form of identity function for saturated proof obligations and does not try to expand them further.

Example 4.1. Recall the judgments $go \vdash^+ \mathsf{g} \mathsf{U} \mathsf{o}$ and $gr \vdash^- \mathsf{g} \mathsf{U} \mathsf{o}$ from Example 3.1, for which we constructed a derivation proof justifying the respectively trace satisfaction/violation as stated in Theorem 1. If we execute the proof-search algorithm of Figure 3 on the initial disjunction set $\{\llparenthesis\,(go, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}$ (corresponding to the proof obligation for $go \vdash^+ \mathsf{g}\ \mathsf{U}\ \mathsf{o}$), the execution terminates, returning the disjunction set $\{\llparenthesis\,\rrparenthesis\}$ *i.e.,* we have no further proof obligations and thus a complete derivation was found. We obtain the same outcome when we run the algorithm on the initial disjunction set for $gr \vdash^- \mathsf{g}\ \mathsf{U}\ \mathsf{o}$, *i.e.,* $\{\llparenthesis\,(gr, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^-\,\rrparenthesis\}$.

Recall also the inconclusive judgement $gg \vdash^+ \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ from Example 3.1. Executing our proof-search algorithm on the initial disjunction set $\{\llparenthesis\,(gg, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}$ terminates with the resultant (saturated) disjunction set $\{\llparenthesis\,\lceil\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o}\rceil^+\,\rrparenthesis\}$. As an illustrative example, we go over this expansion in detail below:

$$
\begin{aligned}
\mathbf{exp}(\{\llparenthesis\,(gg, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}) &= \mathbf{exp}(\{\llparenthesis\,(gg, \mathsf{o})^+\,\rrparenthesis, \llparenthesis\,(gg, \mathsf{g})^+, (g, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}) && (17)\\
&= \mathbf{exp}(\{\} \cup (\{\llparenthesis\,\rrparenthesis\} \oplus \{\llparenthesis\,(g, \mathsf{o})^+\,\rrparenthesis, \llparenthesis\,(g, \mathsf{g})^+, (\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}) && (18)\\
&= \mathbf{exp}(\{\llparenthesis\,(g, \mathsf{o})^+\rrparenthesis, \llparenthesis\,(g, \mathsf{g})^+, (\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}) && (19)\\
&= \{\llparenthesis\,\lceil\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o}\rceil^+\,\rrparenthesis\} && (20)
\end{aligned}
$$

Expansion (17) is obtained by applying the function $\mathbf{expO}(-)$ on the only obligation $(gg, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+$; it returns the obligations derived from the premises of the two proof rules from Figure 2 that may be applied, namely pUnt1 and pUnt2. At (18) the two conjunction sets are expanded: $\mathbf{expC}(\llparenthesis\,(gg, \mathsf{o})^+\,\rrparenthesis)$ yields $\{\}$ (terminating unsuccessfully the proof search along this potential derivation), whereas $\mathbf{expC}(\llparenthesis\,(gg, \mathsf{g})^+, (g, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis)$ returns $\{\llparenthesis\,\rrparenthesis\}$ from $\mathbf{expO}((gg, \mathsf{g})^+)$ and $\{\llparenthesis\,(g, \mathsf{o})^+\,\rrparenthesis, \llparenthesis\,(g, \mathsf{g})^+, (\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+\,\rrparenthesis\}$ from $\mathbf{expO}((g, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+)$—similar to the expansion in (17). Merging these sets using $\oplus$ yields (19). Its expansion follows a similar pattern to that of (18), with the exception that $\mathbf{expO}((\epsilon, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^+)$ returns a saturated set, at which point the expansion terminates.

Finally, recall the failed derivation for the judgement $go \vdash^- \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ from Example 3.1, where we argued that the reason why a derivation could not be constructed in this case was qualitatively different from that of the judgement $gg \vdash^+ \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ discussed above—no extension of the partial trace go could every lead to a completed satisfaction. Accordingly, expanding the initial disjunction set $\{\llparenthesis\,(go, \mathsf{g}\,\mathsf{U}\,\mathsf{o})^-\,\rrparenthesis\}$ corresponding to the judgement $go \vdash^- \mathsf{g}\ \mathsf{U}\ \mathsf{o}$ terminates, but yields a different outcome from the expansion detailed above: it returns the disjunction set $\{\}$. ■

4.2. Properties of the Algorithm

An execution of $\mathbf{exp}(\{\llparenthesis\,(t, \varphi)^q\,\rrparenthesis\})$ may yield either of *three* verdicts. Apart from *success*, $\{\llparenthesis\,\rrparenthesis\}$, meaning that a full proof tree was derived, the algorithm partitions negative results as either a definite *fail*, $\{\}$, or an *inconclusive* verdict, consisting of a saturated disjunction set d (where $c \in d$ implies $\mathbf{sat}(c)$).

Lemma 1. *$\boldsymbol{exp}(d) = d'$ implies $d'=\{\}$ or $d'=\{\llparenthesis\,\rrparenthesis\}$ or ($c \in d$ implies $\boldsymbol{sat}(c)$).*

Proof. Immediate from the definition of $\mathbf{exp}(d)$ in Figure 3. □

The algorithm of Figure 3 for the proof rules of Figure 2 shows that the function $\mathbf{exp}(d)$ is *decidable* for the three outcomes discussed above (*i.e.,* success, failure or inconclusive verdicts). Intuitively, the main reason for this is because the proof system is *cut-free*, where rule premises are either defined in terms of string suffixes or sub-formulas. Formally, we define a rank function $|-|$ mapping proof obligations to pairs of naturals, for which we assume a lexicographical ordering $(n_1, m_1) \geq (n_2, m_2) \stackrel{\text{def}}{=} n_1 > n_2 \vee (n_1 = n_2 \wedge m_1 \geq m_2)$ and the obvious function $\mathbf{max}(-)$ returning the greatest element from a set of such pairs. Apart from $|t|$, we also assume $|\varphi|$ returning the *maximal depth* of the formula (*e.g.,* $|p \,\mathsf{U}\, \neg(\bar{p} \vee p)| = 4$ and $|\bar{p}| = 1 = |\mathsf{tt}|$).

$$|(t,\varphi)^q| \stackrel{\text{def}}{=} (|t|, |\varphi|) \qquad |c| \stackrel{\text{def}}{=} \mathbf{max}(\{|o| \mid o \in c\} \cup \{(0,0)\})$$
$$|\lceil t,\varphi\rceil^q| \stackrel{\text{def}}{=} (0,0) \qquad |d| \stackrel{\text{def}}{=} \mathbf{max}(\{|c| \mid c \in d\} \cup \{(0,0)\})$$

Above, the rank function maps saturated obligations to the bottom element $(0, 0)$. We overload the function to conjunction sets, where we add $(0, 0)$ to the $\mathbf{max}(-)$ calculation to cater for the case where c is empty. Following a similar pattern, we also extend the rank function to disjunction sets, but equate all sets with an empty conjunction set to the bottom element $(0, 0)$; this mirrors the termination condition of the algorithm in Fig. 3 which terminates the search as soon as the shortest proof tree is detected.

We can show two important properties. The first one, Lemma 2, states that when the disjunction set rank is $(0, 0)$, then expanding it is just an idempotent operation. The second property, Lemma 3, states that if a disjunction set d has a rank that is not $(0, 0)$, then expanding each of its constituent conjunction sets yields a disjunction set with rank that is strictly less than that of d. Since each iteration of an expansion of d is defined as the union of all the respectively expansions of its constituent conjunction sets (see Figure 3), this means that at each iteration the rank of the disjunction set will strictly decrease. This carries on until it reached the only rank that cannot decrease further, *i.e.,* $(0, 0)$, at which point the expansion must terminate (by Lemma 2).

Lemma 2. $|d| = (0, 0)$ *implies* $\boldsymbol{exp}(d) = d$

Proof. Immediate from the definition of $|d|$, where we have $|d| = (0, 0)$ only when $d = \{\}$ or $d = \{(\!|\,|\!)\}$ or else ($c \in d$ implies $\mathbf{sat}(c)$). □

Lemma 3. $|d| \neq (0, 0)$ *implies* $|d| > |\bigcup_{c \in d} \boldsymbol{expC}(c)|$

Proof. Follows from showing that $|o| \neq (0, 0)$ implies $|o| > |\mathbf{expO}(o)|$. Since $|o| \neq (0, 0)$ then it must be the case that $o = (t, \varphi)^q$ for some t, φ and q. The proof proceeds by a (long and tedious) case analysis of φ and q. For instance, when $q = +$ and $\varphi = \varphi_1 \mathsf{U} \varphi_2$, $\mathbf{expO}(o)$ at most returns a disjunction set with conjunction sets containing either of the following obligations:

- $\lceil t, \varphi\rceil^q$: $|\lceil t, \varphi\rceil^q| = (0, 0)$ which is clearly less than $|(t, \varphi)^q|$.
- $(t, \varphi_2)^q$: $|(t, \varphi_2)^q|$ is strictly less than $(t, \varphi_1 \mathsf{U} \varphi_2)^q$ since $|\varphi_1 \mathsf{U} \varphi_2| > |\varphi_2|$.

- $(t, \varphi_1)^q$: $|(t, \varphi_1)^q|$ is strictly less than $(t, \varphi_1 \mathsf{U} \varphi_2)^q$ since $|\varphi_1 \mathsf{U} \varphi_2| > |\varphi_1|$.
- $(t', \varphi)^q$ for some t', σ where $t = \sigma t'$: since $|t| > |t'|$ we have $(t, \varphi)^q > (t', \varphi)^q$ again. $\square$

Theorem 3 (Decidability)**.** *For any t, φ, q $\boldsymbol{exp}(\{\llparenthesis (t, \varphi)^q \rrparenthesis\})$ is decidable.*

Proof. Termination follows from Lemma 2 and Lemma 3. Decidability is obtained from termination and Lemma 1. $\square$

Saturated disjunction sets make the algorithm *incremental*, in the following sense. When a further suffix t' is obtained, in addition to a judgement $t \vdash^q \varphi$ with an inconclusive verdict, we can *reuse* the saturated disjunction set returned for $t \vdash^q \varphi$, instead of processing $tt' \vdash^q \varphi$ from scratch. This is done by converting each obligation of the form $\lceil \epsilon, \varphi \rceil^q$ in each saturated conjunction set to the *active* obligation $(t', \varphi)^q$ by extending the auxiliary append function $\mathbf{app}(-)$ defined earlier to disjoint sets:

$$\mathbf{app}(d, t) \stackrel{\text{def}}{=} \{\mathbf{app}(c, t) \mid c \in d\}$$

Example 4.2. To determine whether $ggo \vdash^+ \mathsf{g} \ \mathsf{U} \ \mathsf{o}$ holds, we can take the inconclusive outcome of $\mathbf{exp}(\{\llparenthesis (gg, \mathsf{g} \, \mathsf{U} \, \mathsf{o})^+ \rrparenthesis\})$ from Example 4.1, convert the saturated obligations using suffix o,

$$\mathbf{app}(\mathbf{exp}(\{\llparenthesis (gg, \mathsf{g} \, \mathsf{U} \, \mathsf{o})^+ \rrparenthesis\}), o) = \mathbf{app}(\{\llparenthesis \lceil \epsilon, \mathsf{g} \, \mathsf{U} \, \mathsf{o} \rceil^+ \rrparenthesis\}, o) = \{\llparenthesis (o, \mathsf{g} \, \mathsf{U} \, \mathsf{o})^+ \rrparenthesis\},$$

and then calculate from that point onwards, $\mathbf{exp}(\{\llparenthesis (o, \mathsf{g} \, \mathsf{U} \, \mathsf{o})^+ \rrparenthesis\}) = \{\llparenthesis \, \rrparenthesis\}$. ■

The property of incrementality can be formalised as Theorem 4 below. Its proof relies on the following technical lemmata. Lemma 4 states that appending to a verdict disjunction set $\{\}$ or $\{\llparenthesis \, \rrparenthesis\}$ ignores the appended trace and leaves the disjunction set unchanged. Lemma 5 states that saturated proof obligations can only be obtained when expanding obligations with an empty string. Lemma 6 states that appending a disjunction set with a trace that can be decomposed into two sub-traces yields the same result as that of appending incrementally the two sub-traces. Finally, Lemma 7 states that when a disjunction set is appended and subsequently expanded, then we can expand that disjunction set before appending and still obtain the same answer.

Lemma 4. *$d = \{\}$ or $d = \{\llparenthesis \, \rrparenthesis\}$ implies $\boldsymbol{app}(d, t) = d$.*

Lemma 5. *$\boldsymbol{expO}(o) = \lceil \epsilon, \varphi \rceil^q$ implies $o = (\epsilon, \varphi)^q$*

Lemma 6. *$\boldsymbol{app}(d, t_1 t_2) = \boldsymbol{app}(\boldsymbol{app}(d, t_1), t_2)$*

Lemma 7. *$\boldsymbol{exp}(\boldsymbol{app}(d, t)) = \boldsymbol{exp}(\boldsymbol{app}(\boldsymbol{exp}(d), t))$*

Theorem 4 (Incrementality)**.**

$$\boldsymbol{exp}(\{\llparenthesis (t_1 t_2, \varphi)^q \rrparenthesis\}) = \boldsymbol{exp}(\boldsymbol{app}(\boldsymbol{exp}(\{\llparenthesis (t_1, \varphi)^q \rrparenthesis\}), t_2))$$

Proof. By induction on the structure of t_2:

$t_2 = \epsilon$: We know that $t_1 t_2 = t_1$ and the result follows if we show that

$$\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}) = \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),\epsilon)).$$

From Theorem 3 we know that $\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\})$ is defined. By Lemma 1 we know that it can either be one of three cases: if it is either $\{\}$ or $\{\llparenthesis\,\rrparenthesis\}$, the required result follows immediately from Lemma 4; the only other possibility is that $\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\})$ yields a saturated set d. Now $\mathbf{app}(d,\epsilon)$ would simply yield a disjunction set d' where every saturated obligation of the form $\lceil\epsilon,\varphi\rceil^q$ in d is converted into the respective active obligation $(\epsilon,\varphi)^q$. From Lemma 5 we can then conclude that $\mathbf{exp}(d') = d$ after one expansion.

$t_2 = \sigma t_3$: By I.H. we know that

$$\mathbf{exp}(\{\llparenthesis\,(t_1\sigma t_3,\varphi)^q\,\rrparenthesis\}) = \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1\sigma,\varphi)^q\,\rrparenthesis\}),t_3)).$$

Thus, the required result would follow (by transitivity) if we are able to show that:

$$\mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),\sigma t_3)) = \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1\sigma,\varphi)^q\,\rrparenthesis\}),t_3))$$

The proof proceeds as follows:

$$\begin{aligned}
&\mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),\sigma t_3)) \\
&= \mathbf{exp}(\underline{\mathbf{app}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),\sigma),t_3)}) && \textit{(Lemma 6)} \\
&= \mathbf{exp}(\mathbf{app}(\underline{\mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),\sigma))},t_3)) && \textit{(Lemma 7)} \\
&= \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\underline{\mathbf{app}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\},\sigma)}),t_3)) && \textit{(Lemma 7)} \\
&= \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\underline{\{\llparenthesis\,(t_1\sigma,\varphi)^q\,\rrparenthesis\}}),t_3)) && \textit{(def. of } \mathbf{app}(-)\textit{)}
\end{aligned}$$ □

We can also show another important property of our algorithm, namely that of verdict irrevocability. As stated in the introduction, this means that if a (proof) success or fail verdict is reached for a specific trace prefix, the algorithm preserves this verdict when more evidence from the execution trace is observed. We can indeed show this property via Theorem 4 and the technical lemmata that led to it.

Theorem 5 (Irrevocability)**.**

Fail: $\boldsymbol{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}) = \{\}$ *implies* $\forall t_2\,.\,\boldsymbol{exp}(\{\llparenthesis\,(t_1t_2,\varphi)^q\,\rrparenthesis\}) = \{\}$

Success: $\boldsymbol{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}) = \{\llparenthesis\,\rrparenthesis\}$ *implies* $\forall t_2\,.\,\boldsymbol{exp}(\{\llparenthesis\,(t_1t_2,\varphi)^q\,\rrparenthesis\}) = \{\llparenthesis\,\rrparenthesis\}$

Proof. We here show the proof for the *fail* case since that for the *success* case is analogous. Assume $\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}) = \{\}$ and pick a trace continuation t_2. From Theorem 4, we know that

$$\mathbf{exp}(\{\llparenthesis\,(t_1t_2,\varphi)^q\,\rrparenthesis\}) = \mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis\,(t_1,\varphi)^q\,\rrparenthesis\}),t_2)). \tag{21}$$

From our assumption, Lemma 4 and then the definition of $\mathbf{exp}(-)$ of Figure 3 we can deduce

$$\mathbf{exp}(\mathbf{app}(\mathbf{exp}(\{\llparenthesis (t_1,\varphi)^q \rrparenthesis\}), t_2)) = \mathbf{exp}(\mathbf{app}(\{\}, t_2)) = \mathbf{exp}(\{\}) = \{\} \quad (22)$$

The required result follows from (21) and (22). □

4.3. Runtime Monitoring

The algorithm presented in Section 4.1 can be used to obtain an instantiation of the abstract monitoring system defined in [5, **Definition** 1]. For each formula φ we automatically obtain a monitor m_φ by executing *both* expressions $\mathbf{exp}(\{\llparenthesis (t,\varphi)^+ \rrparenthesis\})$ and $\mathbf{exp}(\{\llparenthesis (t,\varphi)^- \rrparenthesis\})$ concurrently for each execution trace t provided:

- the monitor would *accept* the trace t and all possible continuations, *i.e.*, judgement $\mathbf{acc}(t, m_\varphi)$ in [5], whenever $\mathbf{exp}(\{\llparenthesis (t,\varphi)^+ \rrparenthesis\}) = \{\llparenthesis\,\rrparenthesis\}$, which means that a proof derivation was found for the judgement $t \vdash^+ \varphi$.
- analogously, the monitor would *reject* the trace t and its extensions, *i.e.*, judgement $\mathbf{rej}(t, m_\varphi)$ in [5], whenever $\mathbf{exp}(\{\llparenthesis (t,\varphi)^- \rrparenthesis\}) = \{\llparenthesis\,\rrparenthesis\}$, since a proof derivation for the judgement $t \vdash^- \varphi$ was found.

By virtue of Theorem 1, we know that $\mathbf{acc}(t, m_\varphi)$ corresponds to $ts \in [\![\varphi]\!]$ and that $\mathbf{rej}(t, m_\varphi)$ corresponds to $ts \notin [\![\varphi]\!]$ for any trace completion s respectively, as required in [5]. Note that the possibility of constructing a proof for *both* $t \vdash^+ \psi$ and $t \vdash^- \psi$, is ruled out by soundness (Theorem 1), which implicitly guarantees that our analysis is consistent (since the semantics is defined in terms of sets). It is however possible that m_φ is unable to construct a proof for either case, $t \vdash^+ \psi$ or $t \vdash^- \psi$. This implies that we do not have enough evidence (*i.e.*, the trace produced thus far) to determine satisfaction or violation; this is often equated to the inconclusive verdict "**?**"; see [12]. It is important to note that the algorithm presented in Section 4 can be easily extended so as to record the rules used at each expansion and keep them as part of each conjunction set of proof obligations. This information would then allow us to recover and reconstruct a proof derivation whenever it terminates its search successfully and report it along with the verdict as a high-level implementation-agnostic *explanation* of how the verdict was reached.

The results of Section 4.2 allow us to go a step further and instrument our monitor in an *online synchronous* fashion [26] with the executing system where

- trace events are observed incrementally (one event at a time);
- the system execution is paused while the monitor analyses each event generated.

This synchronous instrumentation relation is used by most RV setups [12], and has been formally specified and extensively studied in [34, 37, 35, 3]. Theorem 4 of Section 4.2 allows us to process a trace prefix of the form $t = \sigma_1 \ldots \sigma_n$

incrementally as the nested sequence $\mathbf{exp}(\mathbf{app}(\ldots\mathbf{exp}(\{(\!|\,(\sigma_1,\varphi)^q\,|\!)\})\ldots,\sigma_n))$ instead of $\mathbf{exp}(\{(\!|\,(t,\varphi)^q\,|\!)\})$. Starting from σ_1, $\mathbf{exp}(\{(\!|\,(\sigma_1,\varphi)^q\,|\!)\})$ for either $q \in \{+,-\}$ can yield either an acceptance, a rejection or (perhaps more importantly) an inconclusive verdict in the form of a saturated disjunction set d_1. This disjunction set d_i represents the internal state of the monitor to be considered when the next event σ_{i+1} is produced by the system as $\mathbf{exp}(\mathbf{app}(d_i,\sigma_{i+1})) = d_{i+1}$. Theorem 5 of Section 4.2 allows us to be efficient and terminate the monitor computation as soon as an acceptance or rejection is reached, safe in the knowledge that this verdict will not change for any of the future events. Finally, Theorem 3 of Section 4.2 provides a guarantee that the monitor cannot interfere with the execution of the system when composed via synchronous instrumentation, because every $\mathbf{exp}(\mathbf{app}(d_i,\sigma_{i+1})) = d_{i+1}$ terminates computing after a finite number of steps. This property is more occasionally referred to as monitor transparency and [34, 33] provides a detailed operational account of how this arises.

5. Alternative RV Symbolic Techniques for LTL

We relate our proof system of Section 3 to two prominent, but substantially distinct, symbolic techniques for LTL in the context of RV, namely Geilen work on informative prefixes [41] and the work on derivatives by Sen *et al.* [61]. In spite of their respectively discrepancies (*e.g.*, [41] works with nnf whereas [61] deals with arbitrary negative formulas, [41] bases analysis on local-informativeness whereas [61] uses a rewriting technique called derivatives) we can use our correspondence results, Theorem 6 and Theorem 7, to better compare these LTL runtime verification techniques to one another.

5.1. Informative Prefixes

Intuitively, an *informative prefix* for a formula *explains* why a trace satisfies that formula [50]. In [41], trace satisfactions are monitored with respect to LTL *formulas in nnf*, by checking whether a trace contains an informative prefix.

Example 5.1. Recall $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ from Example 2.1. Prefix go is informative because

(i) although the head, g, does not satisfy $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ in a definite manner, it allows the possibility of its suffix to satisfy the formula conclusively ($\mathsf{g}(g)$ holds);

(ii) the immediate suffix, o, satisfies $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ conclusively ($\mathsf{o}(o)$ holds).

In [41], both go and o are deemed to be *locally-informative* with respect to $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$ but go generates satisfaction obligations for the immediate suffix (*temporal informative successor*). ■

The algorithm in [41] formalises the notion of locally informative by converting formulas to their *informative normal forms.* Moreover, temporal informative successors are formalised through the function *next*(−), returning a set of formulas from a given formula and a trace element. For instance, in Example 5.1

$$\textsc{gTru}\frac{}{\mathit{linf}(t,\mathsf{tt},\emptyset)} \qquad \textsc{gPre1}\frac{p(\sigma)}{\mathit{linf}(\sigma t,p,\emptyset)} \qquad \textsc{gPre2}\frac{\bar{p}(\sigma)}{\mathit{linf}(\sigma t,\bar{p},\emptyset)}$$

$$\textsc{gOr1}\frac{\mathit{linf}(t,\varphi_1,m)}{\mathit{linf}(t,\varphi_1\vee\varphi_2,m)} \qquad \textsc{gOr2}\frac{\mathit{linf}(t,\varphi_2,m)}{\mathit{linf}(t,\varphi_1\vee\varphi_2,m)}$$

$$\textsc{gAnd}\frac{\mathit{linf}(t,\varphi_1,m_1) \quad \mathit{linf}(t,\varphi_2,m_2)}{\mathit{linf}(t,\varphi_1\wedge\varphi_2,m_1\cup m_2)} \qquad \textsc{gNxt}\frac{}{\mathit{linf}(t,\mathsf{X}\varphi,\{\varphi\})}$$

$$\textsc{gUnt1}\frac{\mathit{linf}(t,\varphi_2,m)}{\mathit{linf}(t,\varphi_1\ \mathsf{U}\ \varphi_2,m)} \qquad \textsc{gUnt2}\frac{\mathit{linf}(t,\varphi_1,m)}{\mathit{linf}(t,\varphi_1\ \mathsf{U}\ \varphi_2,m\cup\{\varphi_1\ \mathsf{U}\ \varphi_2\})}$$

$$\textsc{gRel1}\frac{\mathit{linf}(t,\varphi_1,m_1) \quad \mathit{linf}(t,\varphi_2,m_2)}{\mathit{linf}(t,\varphi_1\ \mathsf{R}\ \varphi_2,m_1\cup m_2)} \qquad \textsc{gRel2}\frac{\mathit{linf}(t,\varphi_2,m)}{\mathit{linf}(t,\varphi_1\ \mathsf{R}\ \varphi_2,m\cup\{\varphi_1\ \mathsf{R}\ \varphi_2\})}$$

Figure 4: Locally-informative and successor judgements

$\mathit{next}(g,\mathsf{g}\ \mathsf{U}\ \mathsf{o})=\{\mathsf{g}\ \mathsf{U}\ \mathsf{o}\}$ whereas $\mathit{next}(o,\mathsf{g}\ \mathsf{U}\ \mathsf{o})=\{\}$. These functions are used in [41] to construct automata that check for these properties over string prefixes.

In this section, we express the locally-informative predicate and the associated temporal informative successors as the single judgement $\mathit{linf}(t,\varphi,m)$, defined as the least relation satisfying the rules in Figure 4. Concretely, a judgement states $\mathit{linf}(t,\varphi,m)$ that t is locally informative for φ with obligations $m\in\mathcal{P}(\textsc{eLtl})$ for the succeeding suffix; recall that eLtl is the extended LTL syntax introduced in Figure 1. For example, for a formula $\mathsf{X}\varphi$, *any* string t is locally informative, but requires the immediate suffix to satisfy φ (see rule gNxt in Figure 4). By contrast, for $\varphi_1\ \mathsf{U}\ \varphi_2$, if t is locally informative for ψ_1 with suffix obligations m, then t is also locally informative for $\varphi_1\ \mathsf{U}\ \varphi_2$ with obligations $m\cup\{\varphi_1\ \mathsf{U}\ \varphi_2\}$ (see rule gUnt2). Informative prefixes are formalised as the predicate $\mathit{inf}(t,\varphi)$ below. Note that recursion in the definition of $\mathit{inf}(t,\varphi)$ is employed on a substring of t and terminates when $m=\emptyset$.

$$\mathit{inf}(t,\varphi)\ \stackrel{\text{def}}{=}\ \exists m.\ \Big(\mathit{linf}(t,\varphi,m) \quad \text{and} \quad (\varphi'\in m \text{ implies } (\mathit{inf}([t]^1,\varphi')))\Big)$$

Example 5.2. We can conclude that go is an informative prefix for $\mathsf{g}\ \mathsf{U}\ \mathsf{o}$, *i.e.*, $\mathit{inf}(go,\mathsf{g}\ \mathsf{U}\ \mathsf{o})$, because we can deduce $\mathit{linf}(go,\mathsf{g}\ \mathsf{U}\ \mathsf{o},\{\mathsf{g}\ \mathsf{U}\ \mathsf{o}\})$ using the rules in Figure 4 and also deduce $\mathit{linf}(o,\mathsf{g}\ \mathsf{U}\ \mathsf{o},\emptyset)$ using the same rules.

$$\textsc{gUnt2}\frac{\textsc{gPre1}\dfrac{\mathsf{g}(g)}{\mathit{linf}(go,\mathsf{g},\emptyset)}}{\mathit{linf}(go,\mathsf{g}\ \mathsf{U}\ \mathsf{o},\{\mathsf{g}\ \mathsf{U}\ \mathsf{o}\})} \qquad \textsc{gUnt1}\frac{\textsc{gPre1}\dfrac{\mathsf{o}(o)}{\mathit{linf}(o,\mathsf{o},\emptyset)}}{\mathit{linf}(o,\mathsf{g}\ \mathsf{U}\ \mathsf{o},\emptyset)}$$

Note that the implication condition required by the definition of $\mathit{inf}(t,\varphi)$ for $\mathit{linf}(o,\mathsf{g}\,\mathsf{U}\,\mathsf{o},\emptyset)$ to be able to infer $\mathit{inf}(o,\mathsf{g}\,\mathsf{U}\,\mathsf{o})$, *i.e.*, $(\varphi'\in\emptyset$ implies $(\mathit{inf}([o]^1,\varphi')))$, is trivially satisfied since there can never be any φ' such that $\varphi'\in\emptyset$. ■

We can formally show a correspondence between informative prefixes and our monitoring proof systems.

Theorem 6 (Informative Prefixes Correspondence)**.** *For all φ in nnf, $\mathit{inf}(t,\varphi)$ iff $t\vdash^{+}\varphi$*

Proof. The *only-if* case is proved by structural induction on φ, then by induction on the structure of t. We here outline the main cases:

$\varphi_1 \wedge \varphi_2$**:** By expanding $\mathit{inf}(t, \varphi_1 \wedge \varphi_2)$, we know that

$$\exists m.\ \mathit{linf}(t, \varphi_1 \wedge \varphi_2, m) \tag{23}$$

$$\varphi' \in m \quad \text{implies} \quad \mathit{inf}([t]^1, \varphi') \tag{24}$$

By case analysis of rules in § 5.1, we have two options to consider for the derivation of $\mathit{linf}(t, \varphi_1 \wedge \varphi_2, m)$ from (23), only the rule GAND could have been used. By this rule, we obtain

$$\mathit{linf}(t, \varphi_1, m_1) \tag{25}$$

$$\mathit{linf}(t, \varphi_2, m_2) \tag{26}$$

$$m = m_1 \cup m_2 \tag{27}$$

Consider that φ'_1 and φ'_2 are in m_1 and m_2 respectively. From (27) we know that $m = m_1 \cup m_2$ and from (24) it follows that

$$\mathit{inf}(t', \varphi'_1) \quad \text{and} \quad \mathit{inf}(t', \varphi'_2) \tag{28}$$

Using (25), (26), and (28) we can obtain

$$\mathit{inf}(t, \varphi_1) \quad \text{and} \quad \mathit{inf}(t, \varphi_2) \tag{29}$$

By (29) and I.H., it follows that

$$t \vdash^+ \varphi_1 \quad \text{and} \quad t \vdash^+ \varphi_2 \tag{30}$$

Finally, by (30) and PAND, we conclude $t \vdash^+ \varphi_1 \wedge \varphi_2$ as required.

$\varphi_1 \mathsf{U} \varphi_2$**:** By expanding $\mathit{inf}(t, \varphi_1 \mathsf{U} \varphi_2)$, we know that

$$\exists m.\ \mathit{linf}(t, \varphi_1 \mathsf{U} \varphi_2, m) \tag{31}$$

$$\varphi' \in m \quad \text{implies} \quad \mathit{inf}([t]^1, \varphi') \tag{32}$$

By case analysis of rules in § 5.1, we have two options to consider for the derivation of $\mathit{linf}(t, \varphi_1 \mathsf{U} \varphi_2, m)$ from (32):

gUnt1: By the rule premise we know that

$$\mathit{linf}(t, \varphi_2, m) \tag{33}$$

By (32) we know that for any arbitrary φ'_2 in m we have

$$\mathit{inf}([t]^1, \varphi'_2) \tag{34}$$

From (33) and (34), we obtain

$$\mathit{inf}(t, \varphi_2) \tag{35}$$

By I.H., it follows that

$$t \vdash^{+} \varphi_2 \tag{36}$$

And finally, by pUnt1, we conclude $t \vdash^{+} \varphi_1 \mathsf{U} \varphi_2$ as required.

gUnt2 By the rule premises we know that

$$\mathit{linf}(t, \varphi_1, m') \tag{37}$$

$$m = m' \cup \{\varphi_1 \mathsf{U} \varphi_2\} \tag{38}$$

From (32), we know that for any arbitrary φ_1' in m' (which is a subset of m) we have

$$\mathit{inf}([t]^1, \varphi_1') \tag{39}$$

By (37) and (39) we obtain $\mathit{inf}(t, \varphi_1)$, and by I.H. we get

$$t \vdash^{+} \varphi_1 \tag{40}$$

From (38) we know that m is not empty, and by (32) we deduce that $[t]^1$ exists. By (32) and (38) it must be the case that

$$\mathit{inf}([t]^1, \varphi_1 \mathsf{U} \varphi_2) \tag{41}$$

By induction on the structure of t, we obtain

$$[t]^1 \vdash^{+} \varphi_1 \mathsf{U} \varphi_2 \tag{42}$$

Finally, from (40), (42) and pUnt2 of Fig. 2 we conclude $t \vdash^{+} \varphi_1 \mathsf{U} \varphi_2$ as required.

The *if* direction is proved by rule induction on $t \vdash^{+} \varphi$ from Figure 2. Note that since φ is in nnf (in our case this means that it does not contain negations) then we shall never require rule pNeg from Figure 2. As a result, our proof does not need to consider properties relating to violation judgements. Again, we here outline the main cases of the inductive proof.

pPrd: From the rule we know that $\varphi = p$. From the rule premises we know $p(\sigma)$. By rule gPre1 of § 5.1, we get $\mathit{linf}(\sigma t, p, \emptyset)$, and given that $m = \emptyset$, it immediately follows that that $\mathit{inf}(\sigma t, p)$ as required.

pAnd: From the rule we know that $\varphi = \varphi_1 \wedge \varphi_2$. From the rule premises $t \vdash^{+} \varphi_1$, $t \vdash^{+} \varphi_2$ and by I.H. we know

$$\mathit{inf}(t, \varphi_1) \tag{43}$$

$$\mathit{inf}(t, \varphi_2) \tag{44}$$

By expanding (43) and (44), we get

$$\mathit{linf}(t, \varphi_1, m_1) \tag{45}$$

$$\varphi_1' \in m_1 \quad \text{implies} \quad \mathit{inf}([t]^1, \varphi_1') \tag{46}$$

$$\mathit{linf}(t, \varphi_2, m_2) \tag{47}$$

$$\varphi_2' \in m_2 \quad \text{implies} \quad \mathit{inf}([t]^1, \varphi_2') \tag{48}$$

Using (45), (47) and the rule gAnd, we obtain

$$\mathit{linf}(t, \varphi_1 \wedge \varphi_2, m_1 \cup m_2) \tag{49}$$

From (46) and (48) we know that for arbitrary φ' in $m_1 \cup m_2$ we have $\mathit{inf}([t]^1, \varphi')$. Thus, by (49) we obtain $\mathit{inf}(t, \varphi_1 \wedge \varphi_2)$ as required.

pUnt1: From the rule we know that $\varphi = \varphi_1 \mathsf{U} \varphi_2$. From the rule premise $t \vdash^+ \varphi_1$ and I.H. we obtain $\mathit{inf}(t, \varphi_1)$ which can be expanded to

$$\mathit{linf}(t, \varphi_1, m) \tag{50}$$

$$\varphi' \in m \quad \text{implies} \quad \mathit{inf}([t]^1, \varphi') \tag{51}$$

Using (50) and the rule gUnt1, we obtain

$$\mathit{linf}(t, \varphi_1 \mathsf{U} \varphi_2, m) \tag{52}$$

From (52) and (51) we can conclude $\mathit{inf}(t, \varphi_1 \mathsf{U} \varphi_2)$ as required.

pUnt2 From the rule we know that $\varphi = \varphi_1 \mathsf{U} \varphi_2$ and that $t = \sigma t'$. From the rule premises $\sigma t' \vdash^+ \varphi_1$, $t' \vdash^+ \varphi_1 \mathsf{U} \varphi_2$ and by I.H. we know

$$\mathit{inf}(\sigma t', \varphi_1) \tag{53}$$

$$\mathit{inf}(t', \varphi_1 \mathsf{U} \varphi_2) \tag{54}$$

By expanding (53) we get

$$\mathit{linf}(\sigma t', \varphi_1, m_1) \tag{55}$$

$$\varphi' \in m_1 \quad \text{implies} \quad \mathit{inf}(t', \varphi') \tag{56}$$

From (55) and rule gUnt2 we obtain

$$\mathit{linf}(\sigma t', \varphi_1 \mathsf{U} \varphi_2, m_1 \cup \{\varphi_1 \mathsf{U} \varphi_2\}) \tag{57}$$

Using $m = m_1 \cup \{\varphi_1 \mathsf{U} \varphi_2\}$ as our witness, to obtain $\mathit{inf}(\sigma t', \varphi_1 \mathsf{U} \varphi_2)$ from (57), we have to show that for any $\varphi' \in m$ we have $\mathit{inf}(t', \varphi')$. This follows from (56) and (54).

pRel1 From the rule we know that $\varphi = \varphi_1 \mathsf{R} \varphi_2$. From the rule premises $t \vdash^+ \varphi_1$, $t \vdash^+ \varphi_2$ and by I.H. we know

$$\mathit{inf}(t, \varphi_1) \tag{58}$$

$$\mathit{inf}(t, \varphi_2) \tag{59}$$

By expanding (58) we get

$$linf(t, \varphi_1, m_1) \tag{60}$$

$$\varphi' \in m_1 \quad \text{implies} \quad inf([t]^1, \varphi') \tag{61}$$

By expanding (59) we get

$$linf(t, \varphi_1, m_2) \tag{62}$$

$$\varphi' \in m_2 \quad \text{implies} \quad inf([t]^1, \varphi') \tag{63}$$

Using (60) and (62) and the rule gRel1, we obtain $linf(t, \varphi_1 \mathsf{U} \varphi_2, m_1 \cup m_2)$. From (5.1), (63) and (5.1) we can conclude $inf(t, \varphi_1 \mathsf{R} \varphi_2)$ as required.

pRel2 From the rule we know that $\varphi = \varphi_1 \mathsf{R} \varphi_2$ and $t = \sigma t'$. From the rule premises $\sigma t' \vdash^+ \varphi_2$, $t' \vdash^+ \varphi_1 \mathsf{R} \varphi_2$ and by I.H. we know

$$inf(\sigma t', \varphi_2) \tag{64}$$

$$inf(t', \varphi_1 \mathsf{R} \varphi_2) \tag{65}$$

By expanding (64) we get

$$linf(\sigma t', \varphi_2, m_1) \tag{66}$$

$$\varphi' \in m_1 \quad \text{implies} \quad inf(t', \varphi') \tag{67}$$

From (66) and rule gRel2 we obtain

$$linf(\sigma t', \varphi_1 \mathsf{R} \varphi_2, m_1 \cup \{\varphi_1 \mathsf{R} \varphi_2\}) \tag{68}$$

Using $m = m_1 \cup \{\varphi_1 \mathsf{R} \varphi_2\}$ as our witness, to obtain $inf(\sigma t', \varphi_1 \mathsf{R} \varphi_2)$, we have to show that for any $\varphi' \in m$ we have $inf(t', \varphi')$. This follows from (67) and (65). □

Discussion. In the *only-if* direction, Theorem 6 embeds informative prefixes within our deductive system, and ensures that our system is as expressive as [41]. In the *if* direction, Theorem 6 shows that every derivation in our proof system corresponds to an informative prefix as defined in [50]; as a corollary, we also establish a correspondence between $t \vdash^- \varphi$ and $inf(t, \neg\varphi)$ as used in [41] for bad prefixes. This provides an alternative justifications as to why our proof system is unable to symbolically process certain prefix and formula pairs, as we illustrate in the next example.

Example 5.3. The proof system of Section 3 is unable to deduce $\epsilon \vdash^+ \mathsf{X}\mathsf{tt}$ (*cf.* proof for Theorem 2) even though, on a semantic level, this holds for any string continuation. Through Theorem 6 we can argue that this is not a mere expressiveness limitation in our proof rules, but because ϵ in *not* an informative prefix of $\mathsf{X}\mathsf{tt}$ as define by Kupferman *et al.* in [50]. ■

Derivative Rewriting Rules

$\mathsf{tt}\{\sigma\} \stackrel{\text{def}}{=} \mathsf{tt}$ $\qquad$ $\mathsf{ff}\{\sigma\} \stackrel{\text{def}}{=} \mathsf{ff}$

$\mathsf{p}\{\sigma\} \stackrel{\text{def}}{=} \textit{if } \mathsf{p}(\sigma) \textit{ then } \mathsf{tt} \textit{ else } \mathsf{ff}$

$(\neg\psi)\{\sigma\} \stackrel{\text{def}}{=} (\mathsf{tt} \oplus \psi)\{\sigma\}$ $\qquad$ $\psi_1 \oplus \psi_2\{\sigma\} \stackrel{\text{def}}{=} \psi_1\{\sigma\} \oplus \psi_2\{\sigma\}$

$\psi_1 \wedge \psi_2\{\sigma\} \stackrel{\text{def}}{=} \psi_1\{\sigma\} \wedge \psi_2\{\sigma\}$ $\qquad$ $\psi_1 \vee \psi_2\{\sigma\} \stackrel{\text{def}}{=} \psi_1\{\sigma\} \vee \psi_2\{\sigma\}$

$\mathsf{X}\psi\{\sigma\} \stackrel{\text{def}}{=} \psi$

$\psi_1 \ \mathsf{U}\ \psi_2\{\sigma\} \stackrel{\text{def}}{=} \psi_2\{\sigma\} \vee (\psi_1\{\sigma\} \wedge \psi_1 \mathsf{U} \psi_2)$

Formula Equivalence Rules

$$\mathsf{tt} \wedge \psi \equiv \psi \qquad \mathsf{ff} \wedge \psi \equiv \mathsf{ff} \qquad \mathsf{ff} \vee \psi \equiv \psi \qquad \mathsf{tt} \vee \psi \equiv \mathsf{tt}$$
$$\psi \wedge \psi \equiv \psi \qquad \psi \vee \psi \equiv \psi \qquad \mathsf{ff} \oplus \psi \equiv \psi \qquad (\psi_1 \wedge \psi_2) \oplus \psi_1 \oplus \psi_2 \equiv \psi_1 \vee \psi_2$$

Figure 5: Derivative Interpretation of LTL formulas

Theorem 6 has another important implication. One argument in favour of limiting symbolic analysis to informative prefixes is that any bad/good prefixes detected can be accompanied by an explanation [19]. However, whereas in [41] this explanation is given in terms of the algorithm implementation, delineating the proof system from its implementing monitor (as in our case) allows us to provide the explanation as a proof derivation using the rules from Section 3, *i.e.,* a reasonably abstract step-by-step justification.

5.2. Derivatives

In a derivatives approach [45, 61], LTL formulas are interpreted as *functions* that take a state *i.e.,* an element of the alphabet Σ, and return another LTL formula. The returned formula is then applied again to the next state in the execution trace, until either one of the *canonical* formulas tt or ff are reached; the trace analysis stops at canonical formulas, since tt (respectively ff) are *idempotent*, returning tt (respectively ff), irrespective of the state that it is applied to. In [61], co-inductive deductive techniques are furthermore used on derivatives to establish LTL formula equivalences, which are then used to obtain optimal monitors for good/bad prefixes.

Example 5.4. Recall $\epsilon \not\vdash^{+} \mathsf{Xtt}$ from Example 5.3. Using their auxiliary co-inductive analysis, in [61] they are able to establish that formulas tt and Xtt are equivalent with respect to good prefixes, $\mathsf{tt} \equiv_G \mathsf{Xtt}$, which allows them to reason symbolically about ϵ and Xtt in terms of ϵ and tt instead. Simply put, using auxiliary reasoning methods, the framework presented in [61] can determine that ϵ suffices to establish that Xtt is satisfied. ■

Formally, a derivative interpretation is expressed as a rewriting operator $\_\{\_\} :: \text{aLTL} \times \Sigma \longrightarrow \text{aLTL}$ (adapted from [61]) defined on the structure of the formula through the rules in Figure 5. In our rule presentation, we position rewriting definitions for core LTL formulas from Figure 1 on the left; formula rewriting however also uses an *extended* set of formulas that include falsehood, ff, disjunction, $\psi_1 \vee \psi_2$, and exclusive-or, $\psi_1 \oplus \psi_2$. The derivatives algorithm also works up to formula normalisations using the equalities presented in Figure 5.

Definition 5.1 (Good/Bad Prefixes [61]). For any finite trace t of the form $\sigma_1\sigma_2 \ldots \sigma_n$:

- t is a *good prefix* for ψ iff $((\psi\{\sigma_1\})\{\sigma_2\})\{\ldots\sigma_n\} \equiv \mathsf{tt}$;
- t is a *bad prefix* for ψ iff $((\psi\{\sigma_1\})\{\sigma_2\})\{\ldots\sigma_n\} \equiv \mathsf{ff}$.

Example 5.5. Recall the LTL formula gUo from Example 2.1. The partial trace *go* is a *good prefix* for gUo according to Definition 5.1 because we can construct the following derivation using the rules in Figure 5:

$$\begin{aligned}
(\mathsf{gUo}\{g\})\{o\} &\stackrel{\text{def}}{=} \mathsf{o}\{g\} \vee \big(\mathsf{g}\{g\} \wedge \mathsf{gUo}\big)\{o\} \\
&\stackrel{\text{def}}{=} \mathsf{ff} \vee \big(\mathsf{g}\{g\} \wedge \mathsf{gUo}\big)\{o\} \;\equiv\; \big(\mathsf{g}\{g\} \wedge \mathsf{gUo}\big)\{o\} \\
&\stackrel{\text{def}}{=} \big(\mathsf{tt} \wedge \mathsf{gUo}\big)\{o\} \;\equiv\; \mathsf{gUo}\{o\} \\
&\stackrel{\text{def}}{=} \mathsf{o}\{o\} \vee \big(\mathsf{g}\{o\} \wedge \mathsf{gUo}\big) \stackrel{\text{def}}{=} \mathsf{tt} \vee \big(\mathsf{g}\{o\} \wedge \mathsf{gUo}\big) \;\equiv\; \mathsf{tt} \qquad \blacksquare
\end{aligned}$$

We can show that good prefixes and bad prefixes, as defined in [61] (reproduced here in Definition 5.1), correspond to finite traces with satisfaction proofs and violation proofs respectively, derived using our proof system of Section 3. Moreover, the correspondence is bidirectional.

Theorem 7 (Derivatives Correspondence). *For any finite trace $t = \sigma_1 \ldots \sigma_n$, and core LTL formula ψ:*

- $(\psi\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt}$ *iff* $t \vdash^{+} \psi$
- $(\psi\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{ff}$ *iff* $t \vdash^{-} \psi$

Proof. For the *only-if* case, we prove both statements simultaneously by induction on length of the string n and then by induction on the structure of ψ. The representative cases are:

$\psi_1 \wedge \psi_2$**:** For the positive case, given that $\psi = \psi_1 \wedge \psi_2$, we have to show that

$$(\psi_1 \wedge \psi_2\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt} \quad \text{implies} \quad t \vdash^{+} \psi_1 \wedge \psi_2$$

From Section 5.2, for $(\psi_1 \wedge \psi_2\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt}$ to hold, then both $(\psi_1\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt}$ and $(\psi_2\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt}$ must also hold. By the I.H. on the structure of ψ, we obtain $t \vdash^{+} \psi_1$ and $t \vdash^{+} \psi_2$. Finally, by rule pAnd, we can conclude that $t \vdash^{+} \psi_1 \wedge \psi_2$ as required. The negative case is analogous.

$\psi_1 \mathsf{~U~} \psi_2$**:** For the positive case we have to show that

$$(\psi_1 \mathsf{~U~} \psi_2\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{tt} \quad \text{implies} \quad t \vdash^{+} \psi_1 \mathsf{~U~} \psi_2$$

From Section 5.2, for $(\psi_1 \mathsf{U} \psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$ to hold, one of the following statements should hold true:

$$(\psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{69}$$
$$(\psi_1\{\sigma_1\} \wedge \psi_1 \mathsf{U} \psi_2)\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{70}$$

- If (69) holds, by I.H. on the structure of ψ, we get $t \vdash^+ \psi_2$ and by rule pUnt1 it follows that $t \vdash^+ \psi_1 \mathsf{U} \psi_2$ as required.
- If (70) holds, then it must be the case that $|\sigma_1 \dots \sigma_n| \geq 1$ and for (70) to hold, it must be the case that

  $$(\psi_1\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{71}$$
  $$(\psi_1 \mathsf{U} \psi_2\{\sigma_2\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{72}$$

  By (71) and I.H. on the structure of the formula we obtain

  $$t \vdash^+ \psi_1 \tag{73}$$

  By (72) and I.H. on the structure of the string we obtain

  $$[t]^1 \vdash^+ \psi_1 \mathsf{U} \psi_2 \tag{74}$$

  By (73), (74) and rule pUnt2 we can conclude that $t \vdash^+ \psi_1 \mathsf{U} \psi_2$ as required.

For the negative case, we have to show that

$$(\psi_1 \mathsf{U} \psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff} \quad \text{implies} \quad t \vdash^- \psi_1 \mathsf{U} \psi_2$$

By the definition given in Section 5.2, for $(\psi_1 \mathsf{U} \psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff}$ to hold, both of the following statements should hold:

$$(\psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff} \tag{75}$$
$$(\psi_1\{\sigma_1\} \wedge \psi_1 \mathsf{U} \psi_2)\{\dots\sigma_n\} \equiv \mathsf{ff} \tag{76}$$

By I.H. on the structure of the formula and (75) we get

$$t \vdash^- \psi_2 \tag{77}$$

From (76), we know that *either* of the following cases must hold:

$$(\psi_1\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff} \qquad \text{or} \tag{78}$$
$$(\psi_1 \mathsf{U} \psi_2\{\sigma_2\})\{\dots\sigma_n\} \equiv \mathsf{ff} \tag{79}$$

We consider either case.

- If (78) holds, then by I.H. on the structure of the formula we obtain $t \vdash^- \psi_1$ and the required result follows by (77) and nUnt1.

- Alternatively, if (79) holds, then we know that $t = \sigma_1 t'$ for some t', and by structural induction on the string we obtain the judgement $t' \vdash^- \psi_1 \mathsf{U}\ \psi_2$. The required result follows by (77) and nUnt2.

$\neg\psi$: For the positive case, we have to show that

$$(\neg\psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt} \quad \text{implies} \quad t \vdash^+ \neg\psi$$

By the definition in Section 5.2, we have $(\mathsf{tt} \oplus \psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$ which is equivalent to

$$\mathsf{tt} \oplus (\psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{80}$$

For (80) to hold, $(\psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff}$ must hold. By I.H. on the structure of the formula, this yields $t \vdash^- \psi$. The required result follows from pNeg. The negative case is analogous.

For the *if* case, we prove both statements simultaneously by rule induction on $t \vdash^+ \psi$ and $t \vdash^- \psi$. The representative cases are:

pAnd: We know that $\psi = \psi_1 \wedge \psi_2$. From the rule premises $t \vdash^+ \psi_1$, $t \vdash^+ \psi_2$ and I.H. we obtain $(\psi_1\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$ and $(\psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$, which imply $(\psi_1 \wedge \psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$.

pUnt2 We know that $\psi = \psi_1 \mathsf{U}\ \psi_2$. From the rule premises $\sigma t' \vdash^+ \psi_1$, $t' \vdash^+ \psi_1 \mathsf{U}\ \psi_2$ and by I.H. we obtain

$$(\psi_1\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{81}$$
$$(\psi_1 \mathsf{U}\ \psi_2\{\sigma_2\})\{\dots\sigma_n\} \equiv \mathsf{tt} \tag{82}$$

From (81), (82) and the definition of the derivatives, the following holds

$$(\psi_1\{\sigma_1\} \wedge \psi_1 \mathsf{U}\ \psi_2)\{\dots\sigma_n\} \equiv \mathsf{tt}$$

which also implies that

$$\psi_2\{\sigma_1\} \vee (\psi_1\{\sigma_1\} \wedge \psi_1 U \psi_2)\{\dots\sigma_n\} \equiv \mathsf{tt}$$

Thus we can conclude that $(\psi_1 \mathsf{U}\ \psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$.

pNeg We know that $\psi = \neg\psi'$. From the rule premise $t \vdash^- \psi'$ and by I.H. we obtain $(\psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff}$ which implies that $(\mathsf{tt} \oplus \psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$ and ultimately (by the definition of Figure 5) that $(\neg\psi\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{tt}$ as required.

nUnt2 We know that $\psi = \psi_1 \mathsf{U}\ \psi_2$. From the rule premises $\sigma t' \vdash^- \psi_2$, $t' \vdash^- \psi_1 \mathsf{U}\ \psi_2$ and I.H. we obtain

$$(\psi_2\{\sigma_1\})\{\dots\sigma_n\} \equiv \mathsf{ff} \tag{83}$$
$$(\psi_1 \mathsf{U}\ \psi_2\{\sigma_2\})\{\dots\sigma_n\} \equiv \mathsf{ff} \tag{84}$$

From (84) and the derivatives definition of Figure 5 we can deduce

$$(\psi_1\{\sigma_1\} \wedge \psi_1 \;\mathsf{U}\; \psi_2)\{\ldots\sigma_n\} \equiv \mathsf{ff} \tag{85}$$

By (83) and (85) we conclude that $\big(\psi_2\{\sigma_1\} \vee (\psi_1\{\sigma_1\} \wedge \psi_1 \;\mathsf{U}\; \psi_2)\big)\{\ldots\sigma_n\} \equiv \mathsf{ff}$ must hold, which yields $(\psi_1 \;\mathsf{U}\; \psi_2\{\sigma_1\})\{\ldots\sigma_n\} \equiv \mathsf{ff}$ by the derivatives definition of Figure 5. □

Discussion. Apart from establishing a one-to-one correspondence between the derivative prefixes and proof deductions for the core LTL formulas in our system, Theorem 7 (together with Theorem 6) allows us to relate indirectly the informative prefixes of Section 5.1 to derivative prefixes, using our proof system as a *unifying framework* to bridge the gap between the two formalisms.

In addition, Theorem 7 allows us to identify from where the additional expressiveness of the analysis in [61] derives from. Specifically, the derivatives formalisation is able to reason about additional satisfactions, such as $\epsilon \in [\![\mathsf{X}\mathsf{tt}]\!]$ from Example 5.4, through the auxiliary deductive systems for formula equivalence with respect to good/bad prefixed, $\vdash \psi_1 \equiv_G \psi_2$ and $\vdash \psi_1 \equiv_B \psi_2$. This opens up the possibility of merging the two approaches, perhaps by extending our deductive system with rules analogous to those shown below, that rely on the auxiliary deductive systems of [61]; one should also consult [3, Section 4.2] for a related discussion on the matter.

$$\textsc{pEq}\,\frac{t \vdash^{+} \varphi_1 \qquad \vdash \varphi_1 \equiv_G \varphi_2}{t \vdash^{+} \varphi_2} \qquad\qquad \textsc{nEq}\,\frac{t \vdash^{-} \varphi_1 \qquad \vdash \varphi_1 \equiv_B \varphi_2}{t \vdash^{-} \varphi_2}$$

6. Conclusion

RV monitors are often constructed as *black boxes*, providing scant explanation to the user on how their verdicts are reached. To address this problem, we presented a proof system that formalises the mechanical reasoning carried out by an online monitor when analysing an execution trace with respect to a correctness property specified as an LTL formula. Proof derivations within this system can then be provided as implementation-agnostic explanations for said verdicts. As opposed to other proof systems for the logic LTL, our deduction system captures closely the constraints encountered in such an online setting (*e.g.*, it is a local proof system, defined over partial traces) while preserving the fact that logic itself is defined over complete traces. We demonstrate that these characteristics indeed reflect online monitoring constraints by presenting an online monitoring algorithm derived from the proof system in a relatively straightforward manner. The concrete contributions of our work are:

1. A *sound*, *local* LTL proof system that infers complete trace inclusion from finite prefixes, Theorem 1, together with an incompleteness result, Theorem 2.

2. A demonstration of the realisability of our approach via mechanisation of proof derivations for this system. We also show its viability in terms of its incrementality, Theorem 4, decidability, Theorem 3, and the production of irrevocable verdicts, Theorem 5.

3. A validation of the expressivity of our proposed approach and an exposition of how the proof system can be used as a unifying framework to relate different runtime monitoring formalisms that are useful for verdict interpretation, Theorem 7 and Theorem 6.

We espouse the methods advocated by a recent body of work [36, 37, 1, 3, 5] and tease apart the specification of the symbolic analysis, *i.e.,* the proof system, from its automation, *i.e.,* the actual monitoring algorithm, which yields a number of advantages. The two-tiered organisation leads to better separation of concerns, and a cleaner organisation that is easier to maintain and understand. For instance, we can localise correctness results leading to a more modular organisation *e.g.,* soundness is determined for the proof system, whereas the decidability of proof search is automation specific. The comparisons with other formal approaches, as shown in Section 5, happens exclusively in terms of the proof system, promoting cross-fertilisation with other symbolic techniques. Once the algorithm determines a satisfaction/violation, it can just present the proof derivation justifying the verdict reached *without detailing how* such a derivation was constructed, *i.e.,* how the proof-search was conducted in the algorithm of Figure 3. The modular organisation also allows us to investigate efficient automation algorithms that lower the overheads of the runtime analysis, while keeping the specification fixed, *i.e.,* the same proof-rules of Figure 2: properties such as Theorem 1 do not need to be recomputed for the new algorithm.

Related Work. Apart from the deductive system for LTL formula equivalence in [61], *i.e.,* $\vdash \varphi_1 \equiv_G \varphi_2$ and $\vdash \varphi_1 \equiv_B \varphi_2$ mentioned briefly in Section 5.2, there are other LTL proof systems specifically developed for LTL [40, 53, 49, 23, 14]. However, each system differs substantially from ours. For instance, the model used in the proof systems of [40, 53] is that of programs, *i.e., sets of traces*, instead of (complete) *individual* traces, as in the case of Figure 1. The work in [40, 49] is concerned with developing tableau methods for inferring the validity of a formula from a conjunction of formulas *i.e.,* their sequents are of the form $\varphi_1, \ldots, \varphi_n \vdash \varphi$ which is considerably different from our (local) proof-system sequents of Figure 2. Similar to [40, 49], the proof system of [23] reasons about the full point-space of LTL formulas, but focussed on studying cut-free sequent systems. In [53], they develop three *tailored* proof systems for separate classes of properties, namely safety, response and reactivity properties. In some sense, our violation proof rules may be seen as rules targetting safety LTL properties (that may be violated over a finite trace [50]), but the technical details of [53] are substantially different from ours. For instance, their classification is based on the syntactic structure of the formulas (*e.g.,* $\mathsf{G}\psi$ in the case of safety properties); this is something our proof rules do not do. Moreover, even though their proof rules include a degree locality by considering all the traces of a *particular program*, they do not consider the constraints pertaining to an RV setting such as deductions from *partial* traces and the inclusion of *violation judgements*. The closest to our work is that of Basin *et al.* [14]. They study a local proof system that is inspired by our proof system with separate judgements for both satisfactions and violations. Again, their setting is different from our since they do not target the concerns of a typical RV setting *e.g.,* they do not work with partial traces and are not concerned with synthesising monitors.

Apart from Manna and Pnueli's LTL definition for *finite (but complete)* traces [54], there is also a substantial body of work that studies alternative LTL semantics for *partial (i.e., incomplete)* traces such as [32, 17, 18, 19, 27]. For instance, in [32], the authors define two mutually-dependent semantics for LTL that approximate to either a satisfaction or a violation respectively whenever a trace is truncated before yielding enough information that allows for a definitive verdict. Although the switching between the approximating semantics when evaluating negation formulas in [32] is reminiscent to the switch between satisfaction and violation judgements in our proof system, our approach *does not* approximate verdicts, as is discussed in Section 3 and further elaborated in Section 4.3. In [19] they propose a three-valued LTL semantics for truncated (finite) traces, and study automata-based monitor constructions with respect to the new semantics; they also use the alternative semantics to give characterisations for (finite) trace properties such as good, bad and ugly prefixes. In [17, 18] they extend this idea and define a four-valued LTL semantics for partial traces, splitting inconclusive verdicts into *temporarily violates* and *temporarily satisfies* verdicts, and show how this semantics facilitates the construction of automata-based monitors. More recently, the authors in [27] even propose a five-valued LTL semantics to deal with the uncertainty of trace-element reordering (in settings without a global-clock) apart from the uncertainty of viewing only part of a complete trace. See [18] for an extensive comparison amongst the various LTL semantics for partial/finite traces. N-values semantics for LTL can be interpreted as specifications for how monitors should behave over partial trace. Although this aspect is related to the semantic interpretation of our proof system, we impose this behavioural specification *at the level of the symbolic analysis*, *i.e.,* the proof system, while leaving the semantics of the logic itself unchanged. This leads to a distinct separation between (logic) satisfactions and violations on the one hand, and (proof-system/monitor) acceptances and rejections on the other.

Explainability has recently garnered more attention in the field of runtime monitoring, taking a variety of forms. There is work that uses monitors to augment verdicts and violating traces to assist fault localisation. For instance, Jia *et al.* [47] and Ahrendt *et al.* [6] employ monitors that do not simply raise a violation, but attribute *blame* to the entity that causes the blame. In the context of Cyber-Physical Systems, Bartocci *et al.* [13] use monitor as part of a testing toolchain to augment traces with information as to which signals violated a property and the time interval in which the properties were violated. The closest to our work, at least in terms of aims, is that of Dawes and Reger [29]. The authors generate context-free grammar representations of rejected traces as a means of explaining the violations detected by their monitors. Their logic, CFTL, describes timed properties, and their explanations also include the severity of the timed-constraint violations using a distance measure. Although similar, LTL is less expressive than their logic but our explanations are more closely tied to the properties satisfied or violated.

Future Work. It would be fruitful to relate other LTL symbolic analyses to the ones discussed in Section 5. Our work may also be used as a point of departure for developing proof systems for other interpretations of LTL. For instance, a different LTL model to that of Section 2 covers both *finite* and infinite traces [5]; this alters the negation propagation identities used for the translation function (*e.g.*, $\neg\mathsf{X}\,\psi \equiv \mathsf{X}\,\neg\psi$ does not hold) and, amongst other things, would require tweaking to the proof rules. Similar issues arise in distributed LTL interpretations such as [16] where instead of having one execution trace, we have a *set of traces* (one for each location). Another avenue for research would be to extend these ideas to other, more expressive logics used in the context RV such as the modal μ-calculus [37, 5]. The work in [5] is particularly relevant to our cause for mechanising the runtime analysis of LTL formulas because it provides connections between monitorability property classes based on operational guarantees and the more traditional monitorability classes such as [7]. Since the modal μ-calculus can embed LTL, results from [5] can also be used to identify syntactic LTL fragments that are both sound and complete in the sense of Theorem 1 and Theorem 2.

We also leave complexity analysis and the assessment of the runtime overheads introduced by our setup for future work. More specifically, the automation proposed in Section 4 is presented merely as a vehicle for demonstrating the proximity of the proof rules to an actual RV monitor. However, one can easily define more efficient proof search algorithms that, for instance, expand common obligations across conjunction sets only once, or circumventing repeated obligation expansions across iterations through techniques such as memoization. In the case of generating higher-level explanations in terms of the proof rules used in a derivation, it may be too expensive to record every rule used. To mitigate this, one could investigate the applicability of the concepts studied in Grigore and Kiefer [43] and make an interesting/uninteresting distinction amongst the relevant rules (*e.g.*, in the case of an $\varphi_1 \mathsf{U} \varphi_2$ formula one would say that rule pUnt1 is relevant, implictly recording only the number of times (as an index) rule pUnt2 is applied before pUnt1 is finally used). More comprehensively, the subject of efficient monitor generation for LTL (and its various logical extensions) in the context of RV has been studied in [63, 2, 4].

References

[1] Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., 2018. A Framework for Parametrized Monitorability, in: Foundations of Software Science

and Computation Structures - 21st International Conference (FOSSACS), Springer. pp. 203–220.

[2] Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö., 2017. On the complexity of determinizing monitors, in: Carayol, A., Nicaud, C. (Eds.), Implementation and Application of Automata - 22nd International Conference, CIAA 2017, Marne-la-Vallée, France, June 27-30, 2017, Proceedings, Springer. pp. 1–13. URL: https://doi.org/10.1007/978-3-319-60134-2_1, doi:10.1007/978-3-319-60134-2_1.

[3] Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K., 2019a. Adventures in monitorability: From branching to linear time and back again. Proceedings of the ACM on Programming Languages (POPL) 3, 52:1–52:29. URL: https://dl.acm.org/citation.cfm?id=3290365.

[4] Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K., 2019b. The cost of monitoring alone, in: Bartocci, E., Cleaveland, R., Grosu, R., Sokolsky, O. (Eds.), From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday, Springer. pp. 259–275. URL: https://doi.org/10.1007/978-3-030-31514-6_15, doi:10.1007/978-3-030-31514-6_15.

[5] Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K., 2019c. An operational guide to monitorability, in: Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, Springer. pp. 433–453. URL: https://doi.org/10.1007/978-3-030-30446-1_23, doi:10.1007/978-3-030-30446-1_23.

[6] Ahrendt, W., Henrio, L., Oortwijn, W., 2019. Who is to blame? runtime verification of distributed objects with active monitors. CoRR abs/1908.10042. URL: http://arxiv.org/abs/1908.10042, arXiv:1908.10042.

[7] Alpern, B., Schneider, F.B., 1987. Recognizing safety and liveness. Distributed Comput. 2, 117–126. URL: https://doi.org/10.1007/BF01782772, doi:10.1007/BF01782772.

[8] Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M.R., Pasareanu, C.S., Rosu, G., Sen, K., Visser, W., Washington, R., 2005. Combining test case generation and runtime verification. Theor. Comput. Sci. 336, 209–234. URL: https://doi.org/10.1016/j.tcs.2004.11.007, doi:10.1016/j.tcs.2004.11.007.

[9] Attard, D.P., Francalanza, A., 2017. Trace partitioning and local monitoring for asynchronous components, in: Cimatti, A., Sirjani, M. (Eds.), Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings, Springer.

pp. 219–235. URL: https://doi.org/10.1007/978-3-319-66197-1_14, doi:10.1007/978-3-319-66197-1_14.

[10] Baier, C., Katoen, J.P., 2008. Principles of Model Checking. Representation and Mind series, MIT Press, Cambridge (MA), USA.

[11] Bartocci, E., Falcone, Y. (Eds.), 2018. Lectures on Runtime Verification - Introductory and Advanced Topics. volume 10457 of *Lecture Notes in Computer Science*. Springer. URL: https://doi.org/10.1007/978-3-319-75632-5, doi:10.1007/978-3-319-75632-5.

[12] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G., 2018. Introduction to runtime verification, in: [11]. pp. 1–33. URL: https://doi.org/10.1007/978-3-319-75632-5_1, doi:10.1007/978-3-319-75632-5_1.

[13] Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Nickovic, D., 2019. Automatic failure explanation in CPS models, in: Ölveczky, P.C., Salaün, G. (Eds.), Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, Springer. pp. 69–86. URL: https://doi.org/10.1007/978-3-030-30446-1_4, doi:10.1007/978-3-030-30446-1_4.

[14] Basin, D.A., Bhatt, B.N., Traytel, D., 2018. Optimal proofs for linear temporal logic on lasso words, in: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings, Springer. pp. 37–55. URL: https://doi.org/10.1007/978-3-030-01090-4_3, doi:10.1007/978-3-030-01090-4_3.

[15] Basin, D.A., Klaedtke, F., Zalinescu, E., 2017. Runtime verification of temporal properties over out-of-order data streams, in: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, Springer. pp. 356–376. URL: https://doi.org/10.1007/978-3-319-63387-9_18, doi:10.1007/978-3-319-63387-9_18.

[16] Bauer, A., Falcone, Y., 2016. Decentralised LTL monitoring. Formal Methods in System Design 48, 46–93. URL: https://doi.org/10.1007/s10703-016-0253-8, doi:10.1007/s10703-016-0253-8.

[17] Bauer, A., Leucker, M., Schallhart, C., 2007. The good, the bad, and the ugly, but how ugly is ugly?, in: Runtime Verification, Springer. pp. 126–138.

[18] Bauer, A., Leucker, M., Schallhart, C., 2010. Comparing LTL Semantics for Runtime Verification. Logic and Computation 20, 651–674.

[19] Bauer, A., Leucker, M., Schallhart, C., 2011. Runtime Verification for LTL and TLTL. Transactions on Software Engineering and Methodology 20, 14.

[20] Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N., 2017. Monitoring networks through multiparty session types. Theor. Comput. Sci. 669, 33–58. URL: `https://doi.org/10.1016/j.tcs.2017.02.009`, doi:`10.1016/j.tcs.2017.02.009`.

[21] Bohlender, D., Köhl, M.A., 2019. Towards a characterization of explainable systems. CoRR abs/1902.03096. URL: `http://arxiv.org/abs/1902.03096`, `arXiv:1902.03096`.

[22] Bradfield, J., Stirling, C., 1992. Local model-checking for infinite state spaces.

[23] Brunnler, K., Lange, M., 2008. Cut-free Sequent Systems for Temporal Logic. Journal of Logic and Algebraic Programming 76, 216 – 225.

[24] Burlò, C.B., Francalanza, A., Scalas, A., 2020. Towards a hybrid verification methodology for communication protocols (short paper), in: Gotsman, A., Sokolova, A. (Eds.), Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings, Springer. pp. 227–235. URL: `https://doi.org/10.1007/978-3-030-50086-3_13`, doi:`10.1007/978-3-030-50086-3\_13`.

[25] Buss, S.R. (Ed.), 1998. Handbook of Proof Theory. Elsevier.

[26] Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A., 2017. A survey of runtime monitoring instrumentation techniques, in: Francalanza, A., Pace, G.J. (Eds.), Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, pp. 15–28. URL: `https://doi.org/10.4204/EPTCS.254.2`, doi:`10.4204/EPTCS.254.2`.

[27] Chai, M., Schlingloff, B., 2014. Online monitoring of distributed systems with a five-valued LTL, in: International Symposium on Multiple-Valued Logic, IEEE. pp. 226–231. URL: `http://dx.doi.org/10.1109/ISMVL.2014.47`, doi:`10.1109/ISMVL.2014.47`.

[28] Cini, C., Francalanza, A., 2015. An LTL Proof System for Runtime Verification, in: TACAS, Springer. pp. 581–595.

[29] Dawes, J.H., Reger, G., 2019. Explaining Violations of Properties in Control-Flow Temporal Logic, in: Runtime Verification - 19th International Conference, RV 2019, Springer. (to appear).

[30] Desai, A., Dreossi, T., Seshia, S.A., 2017. Combining model checking and runtime verification for safe robotics, in: Lahiri, S.K., Reger, G. (Eds.), Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings, Springer. pp. 172–189.

URL: https://doi.org/10.1007/978-3-319-67531-2_11, doi:10.1007/978-3-319-67531-2_11.

[31] Edwards, L., Veale, M., 2017. Slave to the algorithm? why a 'right to an explanation' is probably not the remedy you are looking for. Duke Law and Technology Review 16, 1–65.

[32] Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V., 2003. Reasoning with temporal logic on truncated paths, in: Computer Aided Verification, Springer. pp. 27–39.

[33] Francalanza, A., . A Theory of Monitors. Information and Computation (to appear).

[34] Francalanza, A., 2016. A Theory of Monitors (Extended Abstract), in: Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS, Eindhoven, The Netherlands, pp. 145–161.

[35] Francalanza, A., 2017. Consistently-detecting monitors, in: 28^{th} International Conference on Concurrency Theory (CONCUR), Schloss Dagstuhl. pp. 8:1–8:19. doi:10.4230/LIPIcs.CONCUR.2017.8.

[36] Francalanza, A., Aceto, L., Achilleos, A., Attard, D.P., Cassar, I., Monica, D.D., Ingólfsdóttir, A., 2017a. A Foundation for Runtime Monitoring, in: Runtime Verification - 17th International Conference, RV 2017, Springer. pp. 8–29. URL: https://doi.org/10.1007/978-3-319-67531-2_2.

[37] Francalanza, A., Aceto, L., Ingólfsdóttir, A., 2017b. Monitorability for the Hennessy-Milner logic with recursion. Formal Methods in System Design 51, 87–116. URL: https://doi.org/10.1007/s10703-017-0273-z.

[38] Francalanza, A., Gauci, A., Pace, G.J., 2013. Distributed system contract monitoring. J. Log. Algebraic Methods Program. 82, 186–215. URL: https://doi.org/10.1016/j.jlap.2013.04.001, doi:10.1016/j.jlap.2013.04.001.

[39] Francalanza, A., Pérez, J.A., Sánchez, C., 2018. Runtime verification for decentralised and distributed systems, in: [11]. pp. 176–210. URL: https://doi.org/10.1007/978-3-319-75632-5_6, doi:10.1007/978-3-319-75632-5_6.

[40] Gabbay, D., Pnueli, A., Shelah, S., Stavi, J., 1980. On the temporal analysis of fairness, in: Principles of Programming Languages, ACM, New York, NY, USA. pp. 163–173. URL: http://doi.acm.org/10.1145/567446.567462, doi:10.1145/567446.567462.

[41] Geilen, M., 2001. On the Construction of Monitors for Temporal Logic Properties, in: Runtime Verification, pp. 181–199.

[42] Gilpin, L.H., Bau, D., Yuan, B.Z., Bajwa, A., Specter, M., Kagal, L., 2018. Explaining explanations: An overview of interpretability of machine learning, in: 2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA), pp. 80–89. doi:10.1109/DSAA.2018.00018.

[43] Grigore, R., Kiefer, S., 2015. Tree buffers, in: Kroening, D., Pasareanu, C.S. (Eds.), Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, Springer. pp. 290–306. URL: https://doi.org/10.1007/978-3-319-21690-4_17, doi:10.1007/978-3-319-21690-4_17.

[44] Havelund, K., Peled, D., 2018. Runtime Verification: From Propositional to First-Order Temporal Logic, in: Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, Springer. pp. 90–112. URL: https://doi.org/10.1007/978-3-030-03769-7_7.

[45] Havelund, K., Rosu, G., 2001. Monitoring Programs using Rewriting, in: Automated Software Engineering, IEEE, Wash., DC, USA. pp. 135–143.

[46] Hinrichs, T.L., Sistla, A.P., Zuck, L.D., 2014. Model check what you can, runtime verify the rest, in: Voronkov, A., Korovina, M.V. (Eds.), HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday. EasyChair. volume 42 of *EPiC Series in Computing*, pp. 234–244. URL: https://easychair.org/publications/paper/tq7.

[47] Jia, L., Gommerstadt, H., Pfenning, F., 2016. Monitors and Blame Assignment for Higher-Order Session Types, in: Bodík, R., Majumdar, R. (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, ACM. pp. 582–594. URL: https://doi.org/10.1145/2837614.2837662, doi:10.1145/2837614.2837662.

[48] Kejstová, K., Rockai, P., Barnat, J., 2017. From model checking to runtime verification and back, in: Lahiri, S.K., Reger, G. (Eds.), Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings, Springer. pp. 225–240. URL: https://doi.org/10.1007/978-3-319-67531-2_14, doi:10.1007/978-3-319-67531-2_14.

[49] Kojima, K., Igarashi, A., 2011. Constructive linear-time temporal logic: Proof systems and kripke semantics. Information and Computation 209, 1491–1503. URL: http://dx.doi.org/10.1016/j.ic.2010.09.008, doi:10.1016/j.ic.2010.09.008.

[50] Kupferman, O., Y. Vardi, M., 2001. Model checking of safety properties. Formal Methods System Design 19, 291–314. URL: http://dx.doi.org/10.1023/A:1011254632723, doi:10.1023/A:1011254632723.

[51] Leucker, M., Schallhart, C., 2009. A Brief Account of Runtime Verification. Journal of Logic and Algebraic Programming 78, 293 – 303. URL: http://www.sciencedirect.com/science/article/pii/S1567832608000775, doi:http://dx.doi.org/10.1016/j.jlap.2008.08.004.

[52] Manna, Z., Pnueli, A., 1991a. Completing the temporal picture. Theoretical Computer Science 83, 97–130. doi:10.1016/0304-3975(91)90041-Y.

[53] Manna, Z., Pnueli, A., 1991b. Completing the Temporal Picture. Theoretical Computer Science 83, 97 – 130. URL: http://www.sciencedirect.com/science/article/pii/030439759190041Y, doi:http://dx.doi.org/10.1016/0304-3975(91)90041-Y.

[54] Manna, Z., Pnueli, A., 1995. Temporal Verification of Reactive Systems: Safety. Springer-Verlag New York, Inc., New York, NY, USA.

[55] Neykova, R., Bocchi, L., Yoshida, N., 2017. Timed runtime monitoring for multiparty conversations. Formal Aspects Comput. 29, 877–910. URL: https://doi.org/10.1007/s00165-017-0420-8, doi:10.1007/s00165-017-0420-8.

[56] Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F., 2018. A session type provider: compile-time API generation of distributed protocols with refinements in f#, in: Dubach, C., Xue, J. (Eds.), Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria, ACM. pp. 128–138. URL: https://doi.org/10.1145/3178372.3179495, doi:10.1145/3178372.3179495.

[57] Pnueli, A., 1977. The Temporal Logic of Programs, in: Symposium on Foundations of Computer Science, IEEE, Wash., DC, USA. pp. 46–57. URL: http://dx.doi.org/10.1109/SFCS.1977.32, doi:10.1109/SFCS.1977.32.

[58] Pnueli, A., Zaks, A., 2006. PSL model checking and run-time verification via testers, in: Misra, J., Nipkow, T., Sekerinski, E. (Eds.), FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Springer. pp. 573–586. URL: https://doi.org/10.1007/11813040_38.

[59] Roşu, G., Havelund, K., 2005. Rewriting-Based Techniques for Runtime Verification. Automated Software Engineering 12, 151–197. URL: http://dx.doi.org/10.1007/s10515-005-6205-y, doi:10.1007/s10515-005-6205-y.

[60] Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A., 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des.

54, 279–335. URL: https://doi.org/10.1007/s10703-019-00337-w, doi:10.1007/s10703-019-00337-w.

[61] Sen, K., Rosu, G., Agha, G., 2003. Generating optimal linear temporal logic monitors by coinduction, in: Advances in Computing Science, Springer. pp. 260–275.

[62] Stirling, C., Walker, D., 1991. Local model-checking in the modal mu-calculus. Theoretical Computer Science 89, 161–177.

[63] Tabakov, D., Vardi, M.Y., 2010. Optimized Temporal Monitors for SystemC, in: Runtime Verification, Springer Berlin Heidelberg. pp. 436–451. URL: http://dx.doi.org/10.1007/978-3-642-16612-9_33, doi:10.1007/978-3-642-16612-9_33.

[64] Troelstra, A.S., Schwichtenberg, H., 2000. Basic Proof Theory. Cambridge Tracts in Theoretical Computer Science. 2 ed., Cambridge University Press, New York, NY, USA. URL: http://books.google.is/books?id=x9x6F_4mUPgC.