Monitorability of Expressive Verdicts

Felipe Gorostiaga[1,2,3] and César Sánchez[1]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid (UPM), Spain
[3] CIFASIS, Argentina
{felipe.gorostiaga,cesar.sanchez}@imdea.org

Abstract. Online runtime verification is a formal dynamic technique that studies how to monitor formal specifications incrementally against an input trace. Often, an observed prefix of a behavior is not enough to emit a definite verdict and the monitor must wait to receive more information. Monitorability classifies the set of properties depending on the feasibility to obtain a verdict after a finite observation. Havelund and Peled [20] classified LTL properties according to whether an observation can be extended to a definite answer.
In this paper we present a framework that extends the classification of Havelund and Peled to verdict domains that are richer than Booleans, obtaining a monitorability setting under which some of the verdicts (but not others) can be discarded after a sequence of observations. We study two instances of this setting, quantitative temporal logics and partially ordered domains for stream runtime verification, and we illustrate using examples the different elements of the taxonomy. Finally, we also consider how assumptions on the set of behaviors can improve monitorability, and how imprecise observations can impair monitorability.

1 Introduction

Runtime verification (RV) is a dynamic formal technique for system reliability that studies how events, emitted from a system under study, adhere to a given formal specification. Runtime verification focuses on two main problems: (1) how to generate a monitor from a given specification, and (2) algorithms that take a monitor and process a sequence of input events produced by the system, typically in a incremental manner, attempting to produce a definite verdict. In this paper we use *behavior* to refer to the trace of the system—that is, one infinite sequence of events that a system can produce—and *observation* as the finite sequence of events that monitor receives.

Static formal verification techniques like model checking [14,28] attempt to prove that every behavior of the system satisfies a given specification. In contrast, in runtime verification monitors must decide based on observations. Runtime verification sacrifices completeness to provide an applicable formal extension of testing and debugging. See [19,26] for surveys on runtime verification and the recent book [4].

Early specification languages studied for runtime verification were based on temporal logics, typically LTL [21,13,6], regular expressions [32], timed regular expressions [2], rules [3], or rewriting [30]. Since monitors only see an observation and not a complete behavior, the semantics of temporal logic must be adapted for finite traces. One solution is to adapt the semantics for finite traces [13] that provide a definite answer upon the "termination" of the trace. Another solution is to give a definite answer only if all the behaviors that extend the observation satisfy the specification (declaring satisfaction), or if all such extensions violate the specification (declaring violation). Otherwise, the monitor can produce a temporary "*I don't know*" verdict [6], with the hope to later refine it into a conclusive verdict. The idea of producing an inconclusive verdict was already introduced in the context of stream runtime verification [11] and later used in variants of LTL for finite traces, like LTLf [12] and MLTL [29].

A basic *soundness* criteria states that monitors should never give a verdict that can be later reverted by an extended observation [7]. However, sound monitors can still switch from an indecisive verdict into a definite verdict. The soundness requirement is semantic, in the sense that it is based on the semantics of the logic itself by considering all possible traces that are compatible with the given observation. Monitors can be formally understood as an implementation of a computational function that maps observations into verdicts [20,33,34] that respects the soundness requirement. Therefore, monitoring algorithms correspond to an incremental execution of the monitor as a function. From this perspective monitorability corresponds to the question of the existence of such a computable function.

One of the first definitions of monitorability, given by Pnueli and Zaks [27], establishes that an LTL property is monitorable after an observation u if there is an observation u' that extends u for which the verdict is definitely a violation or there is an observation u' that is an extension of u for which the verdict is a satisfaction. There are properties that are always monitorable for violation, in the sense that every violating behavior has a finite prefix (observation) that is sufficient to determine the violation. For a second class of properties this witness only exists for some behaviors, and for the rest of the properties there is never such a witness observation (these definitions are analogous replacing violation by satisfaction). Havelund and Peled present in [20] a complete taxonomy for LTL, introducing the terms AFR (always finitely refutable), SFR (sometimes finitely refutable) and NFR (never finitely refutable). Their counterparts for a satisfaction verdict are AFS, SFS and NFS. In this paper we study extensions of this taxonomy for more expressive (non-Boolean) verdicts.

It is useful for specification engineers to have very expressive logics to define their properties, but additional expressiveness usually comes at the price of higher complexity in the decision problems and more inefficient algorithms. Since the early languages used in RV were borrowed from static verification where decidability is crucial, these languages only allowed Boolean verdicts. However, runtime verification solves a simpler problem than model-checking so some researchers have been extending the expressivity of RV specification languages.

Examples include logics that can quantify over the data in the events [20,5], extensions of automata with the ability to store and compare data [9], and quantitative semantics for temporal logics [15]. Another direction to extend the expressivity of monitors is Stream Runtime Verification [11,31,16,18,10] that abstract the data used in the monitoring algorithms in temporal logics to arbitrary data. In this paper we extend the Havelund and Peled notions of monitorability to the setting of richer verdicts by studying whether a subset of the possible verdicts can be discarded after witnessing a finite trace. In [12] the monitorability necessarily refers to the ability to give a conclusive verdict after a finite observation, but the logics we consider are defined over infinite traces. In contrast, LTLf [12] and similar logics are interpreted over finite traces. Also, logics that guarantee that verdicts are obtained after a finite number of steps (by the semantics of the logic or some assumption on the input trace), like MLTL [29], are immediately in AFS and AFR.

The standard monitoring studies monitors that are correct for any system under observation, which is considered unknown during the generation of the monitor. However, one can often monitor more effectively for particular systems or under *assumptions* about what the system can do. For example, [36] improves LTL monitoring using a model of the system to prune the set of possible future observations, and [33] considers how to improve the monitoring of hyperproperties using approximations of the system. Similarly, [22] illustrates properties that are not monitorable but become monitorable if one assumes that the input observation satisfies a given LTL formula. In practice, the events obtained from the system may not be perfect, which can affect the monitoring. For example, in [25] the authors study the possibility that events or event values are unknown, so the monitor must deal with the set of possible observations, therefore emitting sets of verdicts. In [23], the authors define the concept of *trace mutations* to capture divergences between observations and behaviors, and study how different mutations affect the monitorability of a property. We present in Section 5 an example of a system and monitoring with richer verdicts that can be monitored under assumptions and event uncertainties, and instantiate the monitorability landscape for the properties monitored. This paves the way for a systematic analysis of monitoring of rich verdicts under assumptions and uncertainties.

In summary, the contributions of the paper are: (1) an extension of the Havelund and Peled taxonomy of monitorability to richer verdicts and in particular to totally and partially ordered domains; and (2) an instantiation of the taxonomy to quantitative temporal logics and to partially ordered domains based on stream runtime verification.

Finally, note that our taxonomy of properties, like the one introduced by Havelund and Peled, is based on the ability of monitors to produce verdicts. Other taxonomies of properties exist. For example, [8] classifies properties based on the use of the temporal operators involved.

The rest of the paper is structured as follows. Section 2 includes the preliminaries. Section 3 introduces the generalization of the monitorability framework to expressive verdicts. This is instantiated to quantitative temporal logics in

Section 4, where the set of verdicts is totally ordered, and to partially ordered domains in Section 5. Finally, Section 6 contains some final remarks.

2 Preliminaries

We use streams (infinite sequences) to represent the *behavior* exhibited by a system. A stream of type D is an infinite sequence of values of D, and we denote the type of the streams of type D as D^ω. We will usually use *record types* to represent the information of different aspects of the system under study. The type $\langle p_0 :: D_0, \ldots, p_n :: D_n\rangle$ represents a record that contains a finite number of entries and assigns a value of type D_i to every variable p_i for $0 \leq i \leq n$. For example, $s \stackrel{\text{def}}{=} (\langle p : \mathit{true}\rangle\ \langle p : \mathit{true}\rangle\ \langle p : \mathit{false}\rangle^\omega) \in \langle p :: \mathit{Bool}\rangle^\omega$ is the stream of $\langle p :: \mathit{Bool}\rangle$ values where p starts with two *true* values and remains *false* thereafter. Given a record value $r \stackrel{\text{def}}{=} \langle p_0 : v_0, \ldots, p_n : v_n\rangle$ we use $r(p_i)$ to refer to v_i for $0 \leq i \leq n$. Given a stream $\sigma \in D^\omega$ and a natural number $i \in \mathbb{N}_0$ we use $\sigma(i)$ to refer to the element of type D at position i in σ. Similarly, we use σ^i to refer to the stream $(\sigma(i)\ \sigma(i+1)\ \ldots)$. For example, $s(0)(p) = \mathit{true}$, $s(50)(p) = \mathit{false}$, and $s^1 = (\langle p : \mathit{true}\rangle\ \langle p : \mathit{false}\rangle^\omega)$.

We use finite sequences to represent *observations* of the behavior of a program. A sequence of type D is a finite sequence of values of D, and we denote the type of the sequences of type D as D^*. The length of a sequence ls is the number of elements in ls, written as $|ls|$. For example, $l \stackrel{\text{def}}{=} [\langle p : \mathit{true}\rangle\ \langle p : \mathit{true}\rangle\ \langle p : \mathit{false}\rangle\ \langle p : \mathit{false}\rangle\ \langle p : \mathit{false}\rangle] \in [\langle p :: \mathit{Bool}\rangle]$ is the stream of assignments of Boolean values to p, which starts with two *true* values and is succeeded by three *false* values. We say that a sequence $ls \in D^*$ of length $|ls| = n$ is a prefix of a stream $s \in D^\omega$ and write $ls \prec s$ if the first n elements of s coincide with the n elements of ls. We also say that s is a continuation of ls. We say that $ls \in D^*$ is a subsequence of a stream $s \in D^\omega$ and write $ls \sqsubset s$ if there is an index i such that $ls \prec s^i$. We also say that s is an expansion of ls. For example, $|l| = 5$, $l \prec s$ (this is, s is a continuation of ls), and obviously $l \sqsubset s$ (this is, s is an expansion of ls). The sequence $[\langle p : \mathit{false}\rangle\ \langle p : \mathit{false}\rangle\ \langle p : \mathit{false}\rangle]$ is also a subsequence of s, because it is a prefix of s^2.

Let $\mathsf{AP} = \{p_0, \ldots, p_n\}$ be a finite set of atomic propositions and $R \stackrel{\text{def}}{=} \langle p_0 :: \mathit{Bool}, \ldots, p_n :: \mathit{Bool}\rangle$ the record type that assigns a Boolean value to each atomic proposition in AP. The syntax of LTL is:

$$\varphi ::= T \mid a \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi\, \mathcal{U}\, \varphi$$

where a is an atomic proposition, $\vee$ and $\neg$ are the usual Boolean disjunction and negation, and $\bigcirc$ and $\mathcal{U}$ are the next and until temporal operators. The semantics of LTL associate behaviors $\sigma \in R^\omega$ with formulas as follows:

$$
\begin{array}{lll}
\sigma \models T & \text{always} & \\
\sigma \models a & \text{iff} & \sigma(0)(a) = \mathit{true} \\
\sigma \models \bigcirc\varphi & \text{iff} & \sigma^1 \models \varphi \\
\sigma \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models \varphi_1\, \mathcal{U}\, \varphi_2 & \text{iff} & \text{for some } j \geq 0\ \ \sigma^j \models \varphi_2, \text{and for all } 0 \leq i < j, \sigma^i \models \varphi_1
\end{array}
$$

Common derived operators are $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\varphi_1 \vee \varphi_2)$, $\varphi_1 \mathcal{R} \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$, $\Diamond\varphi \stackrel{\text{def}}{=} T\,\mathcal{U}\,\varphi$ and $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$.

2.1 LTL property classification

In [20], the authors give a property classification according to the capability of a monitor to reach a verdict witnessing a finite trace. The original definitions are the following. For a given property φ:

Safety/Always Finitely Refutable (AFR). When φ does not hold on a behavior, a failed verdict can be identified after a finite prefix.

Guarantee/Always Finitely Satisfiable (AFS). When φ is satisfied on a behavior, a satisfied verdict can be identified after a finite prefix.

Liveness/Never Finitely Refutable (NFR). When φ does not hold on a behavior, a refutation can not be identified after a finite prefix.

Morbidity/Never Finitely Satisfiable (NFS). When φ is satisfied on a behavior, satisfaction can not be identified after a finite prefix.

The authors define two extra property classes that are not given a name:

Sometimes Finitely Refutable (SFR). For some behaviors that violate φ, a refutation can be identified after a finite prefix; while for other behaviors violating φ, a refutation cannot be identified with a finite prefix.

Sometimes Finitely Satisfiable (SFS). For some behaviors that satisfy φ, satisfaction can be identified after a finite prefix; while for other behaviors satisfying the property, satisfaction cannot be identified with a finite prefix.

Fig. 1 shows the landscape of property classes along with an example LTL property for every class.

We can see, for example, that $\Diamond p$ belongs to **Guarantee** and **Liveness**. This property is NFR because given any finite prefix of a trace where the property does not hold, we can construct an alternative continuation where it does hold, simply making the next value of p be *true*. The property is also AFS because we can find the first index when p becomes *true* and any continuation of that prefix makes the property *true*.

3 A Richer View of Monitorability

In this section we generalize the framework of Havelund and Peled in [20] to consider richer verdicts (beyond Boolean values). Similar to the approach in [20], we base our work on the ability of a monitor to reach a verdict witnessing a finite observation. Note that the finite satisfiability of a property means that with a finite observation we can *dismiss* the value *false* as the result, and the finite refutability of a property means that with a finite observation we can *dismiss* the value *true* as the result. The main intuition is to focus on the dismissibility of verdict values.

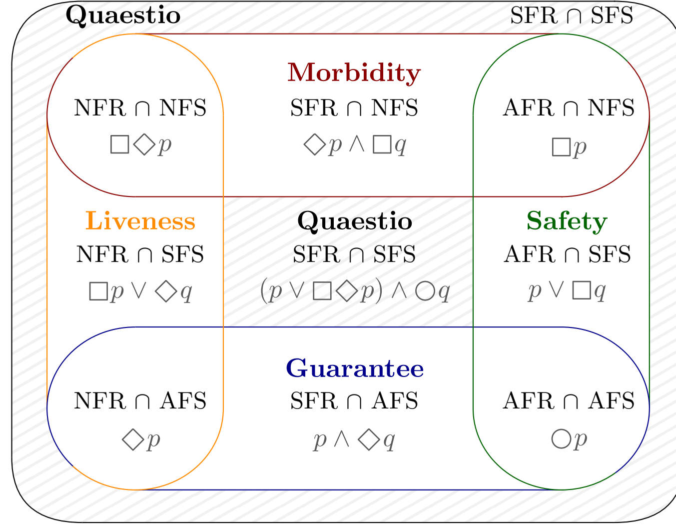

Fig. 1. Landscape of property classes according to [20].

Consider a formalism whose semantics $[\![\cdot]\!]$ is defined over behaviors of type I^ω and that assigns verdicts of type D. For example, classical LTL is defined over records of Boolean values and its semantics assigns Boolean verdicts.

We say a value $v \in D$ is *Finitely Dismissible* for a formula φ and a behavior $s \in I^\omega$ if there is an observation $ls \in I^*$, $ls \prec s$ such that for all s' continuation of ls, $[\![\varphi]\!](s') \neq v$. We say a value $v \in D$ is *Finitely Admissible* for a formula φ and a behavior $s \in I^\omega$ if there is an observation $ls \in I^*$, $ls \prec s$ such that for all possible continuations $s' \in I^\omega$ (this is, all the streams $s' \in I^\omega$ such that $ls \prec s'$), $[\![\varphi]\!](s') = v$. Notice that the only value that can be Finitely Admissible for φ over s is $[\![\varphi]\!](s)$.

We say that a set of values $D' \subseteq D$ is *None Finitely Dismissible* (NFD) for a formula φ and a behavior s if every $v \in D'$ is not Finitely Dismissible for φ and s. Analogously, we say that a set of values $D' \subset D$ is *All Finitely Dismissible* (AFD) for a formula φ and a behavior s if every $v \in D'$ is Finitely Dismissible for φ and s. Notice that the empty set is both NFD and AFD. We say that a set of values $D' \subset D$ is *Some Finitely Dismissible* (SFD) if it is not AFD nor NFD.

We can extend the definition of Finite Admissibility to sets of values but they are of little use in our work.

Lemma 1. *If v is Finitely Admissible for a formula φ and a behavior s then $D \setminus \{v\}$ is AFD for φ and s.*

Proof. Since v is Finitely Admissible for φ and s, there is a finite sequence $ls \prec s$ such that for every continuation s' of ls, $[\![\varphi]\!](s) = v$. We can therefore dismiss any value in $D \setminus \{v\}$ with the finite prefix ls. □

The converse holds for finite domains.

Lemma 2. *If $D \setminus \{v\}$ is AFD for a formula φ and a behavior s and D is finite, then v is Finitely Admissible for φ and s.*

Proof. There is an index for every element v' in $D \setminus \{v\}$ that indicates the shortest length of the finite prefix after which v' can be dismissed for φ over s. After a prefix of the maximum length of those indexes (which are finite), we will have dismissed every $v' \neq v$ in D, and as a consequence the semantics of any continuation over φ is v. □

However, if D is infinite, Lemma 2 does not hold.

Lemma 3. *If $D \setminus \{v\}$ is AFD for a formula φ and a behavior s and D is infinite, then it is not necessarily the case that v is Finitely Admissible for φ and s.*

Proof. Let there be a property φ that assigns the maximum value of the field p (of type $\mathbb{N}$) in the behavior if it exists, and ∞ otherwise. The verdict is of type $\mathbb{N} \cup \{\infty\}$ and for the behavior $s \stackrel{\text{def}}{=} (\langle p : 1\rangle\ \langle p : 2\rangle\ \langle p : 3\rangle\ \ldots)$, the semantics of φ is $[\![\varphi]\!](s) = \infty$, any natural number is finitely dismissible and yet ∞ is not finitely admissible: we can simply repeat the last value of a prefix forever, creating a continuation whose semantics over φ is a natural number. □

We will show two more (counter) examples for bounded, dense verdict domains in Sections 4.2 and 4.3.

4 Boolean and Quantitative Totally Ordered Domains

In this section we generalize the classification of Havelund and Peled to totally ordered sets, according to the dismissibility of values with respect to the result. Note that this is the same criterion as in the original definitions.

4.1 Property classes

If the type D of the verdicts of a formalism is a totally ordered set equipped with an order relation $(D, \leq)$, we can classify the properties according to their value-dismissibility as follows. Let $v = [\![\varphi]\!](\sigma)$ be the semantics of the property φ for behavior σ. We use $v_<$ for the set of values lower than v and and $v_>$ the set of values greater than v, that is $v_< \stackrel{\text{def}}{=} \{v' | v' < v\}$ and $v_> \stackrel{\text{def}}{=} \{v' | v' > v\}$. We say a property is AFD$_>$ if the set of values greater than its verdict for any behavior is AFD. We define AFD$_<$, NFD$_>$ and NFD$_<$ analogously. A property is SFD$_>$ if for some executions, some values greater than its verdict are finitely dismissible while other are not. The definition of SFD$_<$ is analogous. With these definitions we can redefine the property classes for rich, totally ordered domains as follows:

Safety/AFD$_>$. We say that a property is a *Safety* property if the set $v_>$ is All Finitely Dismissible for any behavior σ (this is, the monitor can dismiss every value greater than the result with a prefix). In other words, if you set a maximum tolerable threshold t and the result is below the threshold, you will know it after a finite prefix.

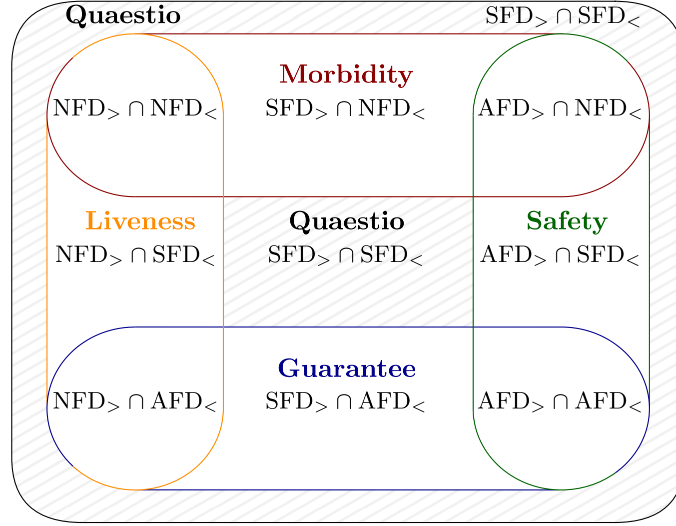

Fig. 2. Landscape of property classes for totally ordered domains

Guarantee/AFD$_<$. We say that a property is a *Guarantee* property if the set $v_<$ is All Finitely Dismissible for any behavior σ (this is, the monitor can dismiss every value lower than the result with a prefix). In other words, if you set a minimum score t and the result is higher than it, you will know it with a finite prefix.

Liveness/NFD$_>$. We say a property is a *Liveness* property if the set $v_>$ is None Finitely Dismissible for any behavior σ (this is, the monitor can never dismiss any value greater than the result processing any prefix). In other words, if you set a maximum tolerable threshold t and the result is below it, you will not know it with a finite prefix.

Morbidity/NFD$_<$. We say a property is a *Morbidity* property if the set $v_<$ is None Finitely Dismissible for any behavior σ (this is, the monitor can never dismiss any value lower than the result with a prefix). In other words, if you set a minimum score t and the result is higher than it, you will not know it with a finite prefix.

We define two additional sets of properties:

SFD$_>$. In some traces the monitor can dismiss some values higher than the result with a prefix, but not others.

SFD$_<$. In some traces, the monitor can dismiss some values lower than the result, but not others.

Fig. 2 shows the landscape of property classes for rich, totally ordered domains. Note that the definitions of **Safety** and **Liveness** are incompatible for verdict domains with more than one element, and so are the definitions of **Guarantee** and **Morbidity**, which means that a property cannot be both a **Safety** and a

Liveness property, nor can it be both a **Guarantee** and a **Morbidity** property. However, it is possible that a property belongs to two classes, and also that a property does not belong to any of the classes described above.

We see that our definitions maintain the classification of the original properties presented in [20] if we consider the Boolean domain with the usual order relation *false* $<$ *true*. Recall that according to our definitions, a **Safety** property is one such that a monitor can always dismiss the values greater than the result with a finite prefix. This is equivalent to say, in the Boolean ordered set, that if the result is *false* then a monitor can always dismiss the set $\{true\}$ with a prefix. Since the domain is finite, Lemma 2 implies that the value *false* is always Finitely Admissible, and thus, a failed verdict can be identified after a finite prefix. A similar analysis can be made for the rest of the classes.

In the following sections we will give a witness for every class and sensible multiclass for different formalisms and domains.

4.2 Quantitative LTL

In [15] the authors define quantitative semantics for LTL, which generalize the semantics from Boolean to a richer type. Input streams are streams of real numbers in the range $[0, 1]$. The syntax is the same as for classic LTL. The semantics is given recursively over the terms and assigns a value in the range $[0, 1]$ for every term with respect to a behavior that assigns a real number in the range $[0, 1]$ to every proposition, this is, in QLTL, $R \stackrel{\text{def}}{=} \langle p_0 :: \mathbb{R}_{[0,1]}, \ldots, p_n :: \mathbb{R}_{[0,1]} \rangle$.

$$
\begin{aligned}
[\![T]\!](\sigma) &\stackrel{\text{def}}{=} 1 & [\![\varphi \vee \psi]\!](\sigma) &\stackrel{\text{def}}{=} [\![\varphi]\!](\sigma) \sqcup [\![\psi]\!](\sigma) \\
[\![a]\!](\sigma) &\stackrel{\text{def}}{=} \sigma(0)(a) & [\![\neg\varphi]\!](\sigma) &\stackrel{\text{def}}{=} 1 - [\![\varphi]\!](\sigma) \\
[\![\bigcirc\varphi]\!](\sigma) &\stackrel{\text{def}}{=} [\![\varphi]\!](\sigma^1) \\
[\![\varphi\, \mathcal{U}\, \psi]\!](\sigma) &\stackrel{\text{def}}{=} sup_{i\geq 0}([\![\varphi]\!](\sigma^0) \sqcap \cdots \sqcap [\![\varphi]\!](\sigma^{i-1}) \sqcap [\![\psi]\!](\sigma^i))
\end{aligned}
$$

Where $x \sqcap y \stackrel{\text{def}}{=} \min(x, y)$ and $x \sqcup y \stackrel{\text{def}}{=} \max(x, y)$.

Following the syntax of the derived operators, their semantics in QLTL are $[\![\varphi \wedge \psi]\!](\sigma) \stackrel{\text{def}}{=} [\![\varphi]\!](\sigma) \sqcap [\![\psi]\!](\sigma)$, $[\![\Diamond\varphi]\!](\sigma) \stackrel{\text{def}}{=} sup_{i\geq 0}[\![\varphi]\!](\sigma^i)$, and $[\![\Box\varphi]\!](\sigma) \stackrel{\text{def}}{=} inf_{i\geq 0}[\![\varphi]\!](\sigma^i)$.

Since the generalization of the property classes to quantitative values presented in Section 4.1 is consistent with the generalization of the semantics of LTL to quantitative values in QLTL, the formulae presented in Section 2.1 belong to the same classes.

Lemma 4. *The following hold:*
- *The property $\Diamond p$ belongs to* **Guarantee** *and* **Liveness***,*
- *the property $\Box p$ belongs to* **Safety** *and* **Morbidity***,*
- *the property $\bigcirc p$ belongs to* **Safety** *and* **Guarantee***,*
- *the property $\Box\Diamond p$ belongs to* **Morbidity** *and* **Liveness***,*
- *the property $p \wedge \Diamond q$ only belongs to* **Guarantee***,*

– *the property* $\Box p \lor \Diamond q$ *only belongs to* **Liveness**,
– *the property* $p \lor \Box q$ *only belongs to* **Safety**, *and*
– *the property* $\Diamond p \land \Box q$ *only belongs to* **Morbidity**.

Proof. We show the proof that $\Diamond p$ belongs to **Guarantee** and **Liveness**. $\Diamond p$ is $\text{NFD}_>$ because given any finite prefix of a trace where the supremum is $v \neq 1$, we can construct an alternative continuation where it is greater than v simply making the next value 1. If the verdict is $v = 1$, the complement set would be trivially NFD. It is $\text{AFD}_<$ because the verdict v is the minimum element greater or equal than the infinite values of p throughout the trace. Let $v' < v$. If no element in $(v', v]$ occur in the trace, then the result would be v'. Otherwise, the occurrence of such value would dismiss v' as a possible result. □

The verdict of a QLTL property is a real number in the range $[0, 1]$, i.e. an infinite set, and thus it is subject to the case where the set of values different from the result is AFD but the result itself is not Finitely Admissible.

Lemma 5. *There is a QLTL property* φ *and a behavior* s *such that* $v = [\![\varphi]\!](s)$ *is not Finitely Admissible, but* $[0, 1] \setminus \{v\}$ *is AFD.*

Proof. Consider the property $(\Diamond p \land q)$ and a behavior s such that, at every instant i, the value of q is $\frac{1}{2}$ and the value of p is $\sum_{n=0}^{i} \frac{1}{4 \times 2^n}$, this is, p produces values closer to $\frac{1}{2}$, but never $\frac{1}{2}$ exactly. Then, in QLTL, $[\![\Diamond p]\!](s) = \frac{1}{2}$, the set $[0, \frac{1}{2}) \cup (\frac{1}{2}, 1]$ is All Finitely Dismissible, but the result $\frac{1}{2}$ is not Finitely Admissible. □

4.3 Discounting in LTL

The temporal logic $\text{Disc}^{\text{LTL}}[\mathcal{D}]$ generalizes LTL by adding discounting temporal operators [1]. According to the authors, the logic is in fact a family of logics, each parameterized by a set $\mathcal{D}$ of discounting functions. A function $\eta : \mathbb{N} \to [0, 1]$ is a *discounting function* if $\lim_{i \to \infty} \eta(i) = 0$, and η is strictly decreasing. Input streams are Boolean, as in classic LTL, but verdicts are real numbers in the range $[0, 1]$.

For a given a set of discounting functions $\mathcal{D}$, the logic $\text{Disc}^{\text{LTL}}[\mathcal{D}]$ adds to LTL the operator $\varphi \, \mathcal{U}_\eta \, \varphi$. The semantics of this logic is given recursively over the terms and assigns a value in the range $[0, 1]$ for every term with respect to a behavior, assigning 0 to an input value of *false* and 1 to an input value of *true*.

$$[\![T]\!](\sigma) \stackrel{\text{def}}{=} 1 \qquad [\![\varphi \lor \psi]\!](\sigma) \stackrel{\text{def}}{=} \max\{[\![\varphi]\!](\sigma), [\![\psi]\!](\sigma)\}$$

$$[\![a]\!](\sigma) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \sigma(0)(a) = \mathit{true} \\ 0 & \text{otherwise} \end{cases} \qquad [\![\neg\varphi]\!](\sigma) \stackrel{\text{def}}{=} 1 - [\![\varphi]\!](\sigma)$$

$$[\![\bigcirc\varphi]\!](\sigma) \stackrel{\text{def}}{=} [\![\varphi]\!](\sigma^1)$$

$$[\![\varphi \, \mathcal{U} \, \psi]\!](\sigma) \stackrel{\text{def}}{=} \sup_{i \geq 0}\{\min\{[\![\psi]\!](\sigma^i), \min_{0 \leq j < i}\{[\![\varphi]\!](\sigma^j)\}\}\}$$

$$[\![\varphi \, \mathcal{U}_\eta \, \psi]\!](\sigma) \stackrel{\text{def}}{=} \sup_{i \geq 0}\{\min\{\eta(i)[\![\psi]\!](\sigma^i), \min_{0 \leq j < i}\{\eta(j)[\![\varphi]\!](\sigma^j)\}\}\}$$

Property classification. The properties in Section 4.2 belong to the same categories, which is reasonable because they do not use discounting functions. For the same reason, and since the observations are Boolean values, the possible values are $\{0,1\}$ and thus the semantics and the property classes coincide with those of classic LTL.

Lemma 6. *Properties of the form $\varphi \,\mathcal{U}_\eta\, \psi$ belong to* **Safety** *and* **Guarantee***.*

Proof. Let $v = \sup_{i\geq 0}\{\min\{\eta(i)[\![\psi]\!](\sigma(i)), \min_{0\leq j<i}\{\eta(j)[\![\varphi]\!](\sigma(j))\}\}\}$ for a trace σ.

Let $v' \in (v,1]$ (if the set $(v,1]$ is empty, it can be trivially dimissed). Since both $[\![\psi]\!]$ and $[\![\varphi]\!]$ are in the range $[0,1]$ at any index, then there is an index k such that $\eta(k) < v'$. After index k and since no value greater than v' ever happened (it would be greater than the verdict and thus the supremum), v' can be dismissed as a verdict.

Let $v' \in [0,v)$ (if the set $[0,v)$ is empty, it can be trivially dimissed). Following the same reasoning, there is an index k after which $\eta(k) < v$. At index k, v is guaranteed to be the result. □

$\text{Disc}^{\text{LTL}}[\mathcal{D}]$ provides us with another example where the set of non-verdicts is All Finitely Dismissible but the correct result is not Finitely Admissible.

Lemma 7. *There is a $Disc^{LTL}[\mathcal{D}]$ property φ and a behavior s such that $v = [\![\varphi]\!](s)$ is not Finitely Admissible, but $[0,1]\setminus\{v\}$ is AFD.*

Proof. Consider a behavior s such that p is always *false*, and the $\text{Disc}^{\text{LTL}}[\mathcal{D}]$ property $\varphi = \Diamond_\eta p(s)$. The temporal operator $\Diamond_\eta$ is defined as $\Diamond_\eta\varphi \stackrel{\text{def}}{=} T\,\mathcal{U}_\eta\,\varphi$. From the semantics of $\mathcal{U}$ and T, we see that

$$\begin{aligned}[\![\varphi]\!](s) &= \sup_{i\geq 0}\{\min\{\eta(i).\sigma(i)(p), \min_{0\leq j<i}\{\eta(j).1)\}\}\} \\ &= \sup_{i\geq 0}\{\min\{\eta(i).0, \min_{0\leq j<i}\{\eta(j))\}\}\} = \sup_{i\geq 0}\{\min_{0\leq j<i}\{\eta(j))\}\}\end{aligned}$$

Then, $[\![\varphi]\!](s) = 0$, the set $(0,1]$ is All Finitely Dismissible, but the result 0 is not Finitely Admissible. □

5 Towards Partially Ordered Domains

In this section we generalize the property classes definitions presented so far to partially ordered domains. We also introduce the notions of *assumptions* (via gray box monitoring) and *imprecise observability* to capture different relations between behaviors and observations, and we see how these notions impact in property classification in a concrete example.

5.1 Property Classes for Partially Ordered Domains

We first generalize the definitions of the property classes presented in Section 4.1. If the type of the verdicts of a formalism is a partially ordered set, we still classify the properties according to their value-dismissibility. Let $v = [\![\varphi]\!](\sigma)$, $v_{\ngeq}$ the set of values in D not greater or equal than v and $v_{\nleq}$ the set of values in D not lower or equal than v, that is $v_{\nleq} \stackrel{\text{def}}{=} \{v' | v' \nleq v\}$ and $v_{\ngeq} \stackrel{\text{def}}{=} \{v' | v' \ngeq v\}$. We now redefine the property classes for rich, partially ordered domains:

Safety/AFD$_{\nleq}$. We say a property is a *Safety* property if the set $v_{\nleq}$ is All Finitely Dismissible for any behavior σ (this is, the monitor can dismiss every value not lower or equal than the result with a prefix).

Guarantee/AFD$_{\ngeq}$. We say a property is a *Guarantee* property if the set $v_{\ngeq}$ is All Finitely Dismissible for any behavior σ (this is, the monitor can dismiss every value not greater than the result with a prefix).

Liveness/NFD$_{\nleq}$. We say a property is a *Liveness* property if the set $v_{\nleq}$ is None Finitely Dismissible for any behavior σ (this is, the monitor can never dismiss any value not lower or equal than the result with a prefix).

Morbidity/NFD$_{\ngeq}$. We say a property is a *Morbidity* property if the set $v_{\ngeq}$ is None Finitely Dismissible for any behavior σ (this is, the monitor can never dismiss any value not greater or equal than the result with a prefix).

Again, we also define two additional sets of properties:

SFD$_{\nleq}$. In some traces the monitor can dismiss some values not lower or equal than the result with a prefix, but not others.

SFD$_{\ngeq}$. In some traces, the monitor can dismiss some values not greater or equal than the result, but not others.

It is easy to see that for totally ordered domains the set of values not greater or equal than a value is equal to the set of values lower than it, hence these new definitions simply extend the classifications presented in Section 4.1 to partially ordered sets. The landscape of the new property classification is the same as the one in Fig. 2, but with the subscripts $>$ replaced by $\nleq$ and the subscripts $<$ replaced by $\ngeq$.

5.2 Gray Box Monitoring (Assumptions)

So far we have considered that any stream of states is a possible behavior of the system, following a black box approach in which the monitor has no information about the conduct of the system. However, trace analysis for value dismissibility must only take into account *plausible* behaviors of the system under scrutiny and thus we can use *assumptions* to limit the set of behaviors contemplated. We call this a *gray box* approach. An assumption, as defined in [22], is a set of behaviors that contains the traces that comply with the assumption. Assumptions can

make properties fit into the categories presented in the previous section that would otherwise be uncategorizable.

For example, we saw in Section 2.1 that the LTL property $\Box p \vee \Diamond q$ only belongs to the **Liveness** property class. However, under the assumption that $\Box(p \vee q)$, the property becomes a tautology and thus, it is trivially both a **Safety** and **Guarantee** property. The same LTL property $\Box p \vee \Diamond q$ becomes a **Safety** property under the assumption that once q becomes *false*, it will remain *false* forever, i.e., $\Box(\neg q \rightarrow \Box \neg q)$.

Recall that a value v is finitely dismissible (resp. admissible) if the semantics of a property for every continuation of an observation is different from (resp. equal to) v. When we use assumptions, we only need to consider the continuations of the observation ls that intersect the assumption A: $\{s' \in I^\omega \cap A | ls \prec s'\}$.

5.3 Imprecise Observations

Sometimes observations are imperfect, in the sense that some parts of the observation are missing. In practice, this could be due to technical impossibility, bad instrumentation, privacy concerns, faulty communication, or because the monitor is incorporated to an already running system. Up to this point we have considered observations to be a prefix of the behavior, but in this section we generalize the relationship between observations and behaviors via an abstraction function *obs*, which indicates the different ways a behavior can be observed.

The observation function is a representation about how a behavior can be perceived by the monitor. The choice of the observation function has an impact on property classification. For example, we saw in Section 2.1 that the LTL property $\bigcirc p$ is a **Safety** and **Guarantee** property, but it becomes a **Liveness** and **Morbidity** property under loss or corruption of events, stuttering, or incorrect event order arrival.

In the example shown in Section 5.4 below, our *obs* function captures the error (mutation in the terminology of [23]) of losing a prefix of the behavior, as a way to represent the scenario in which we start to monitor a system that is already running and thus the initial state is unknown. We also show a set of *obs* functions that implement a *controlled corruption* mutation in which events of a certain kind are replaced by a *no-value* event, representing the situation in which the system under analysis is not properly instrumented or privacy concerns prevent the monitor from detecting specific events, and thus some events are unobservable.

5.4 Example: Resource Sharing

This example illustrates that if assumptions are present, sometimes it is possible to effectively monitor liveness and safety properties. The monitors considered in this example try to compute a verdict value at every time instant, instead of a single verdict corresponding to the valuation of a formula at the initial position for the input trace observed. We model these streams of valuations in stream

runtime verification [31] where the output of the verdict stream provides the sequence of verdict values for each time instant.

In the scenarios presented so far, the monitor gains information about the behavior by incrementally observing a prefix of the behavior. That is, the monitor observes the set of finite prefixes of a trace, which acts as the abstraction function of a behavior. In other words, the observations of the monitor is computed from a behavior s as $obs_{\prec}(s) \stackrel{\text{def}}{=} \{ls \mid ls \prec s\}$. However, property classification can also be applied to scenarios where the beginning of a trace is unknown, that is, where observations miss a prefix. We can represent this case, considering that observations $ls \in D^*$ are not prefixes of behaviors $s \in D^\omega$ but subsequences of them. The behavior abstraction we consider in this situation is the function that returns each of the (finite) subsequences of a stream, this is, $obs_{\sqsubset}(s) \stackrel{\text{def}}{=} \{ls \mid ls \sqsubset s\}$.

Consider for example a monitor that observes the lock/free operations of semaphores of a concurrent program. Our task is to study a monitor that produces a verdict indicating, at every point in time and for every resource, which process is the holder of the lock (if any). We start by introducing some intermediate definitions and properties before we describe the monitor.

The input stream $e \in EventT^\omega$ indicates the successive events that take place during the execution of the concurrent program. In [17] the authors show how to represent event-based systems using a synchronous language. As proposed in that work, we use a special constant *nop* to represent the absence of an event in an instant. We assume that at most one event can happen at every instant. Let *ProcessT* be the set of processes in the system, and let *ResourceT* be the set of resources. A process can *lock* or *free* a resource.

The output (verdict) stream $acquired \in AcquiredT^\omega$ is calculated from e and computes which process is holding which resource at every instant keeping a map that assigns a process to a resource. Formally, the types *EventT* and *AcquiredT* are defined as:

$$\begin{aligned} EventT &\stackrel{\text{def}}{=} \{nop\} \cup (ProcessT \times ResourceT \times \{lock, free\}) \\ AcquiredT &\stackrel{\text{def}}{=} ResourceT \mapsto ProcessT \end{aligned}$$

A *nop* event represents that no event happened in an instant. A (p, r, o) event indicates that process p has performed operation o over resource r. The resources that are not a key of the map in *AcquiredT* are unlocked, and we define maps as sets of key-value pairs with at most one value associated with a key.

We define a partial order relation between maps: for two maps m and m', we say that $m \leq m'$ iff every key-value pair in m is in m'. Formally, $m \leq m' \stackrel{\text{def}}{=} \forall (k, v) \in m, (k, v) \in m'$. For example, the empty map $\emptyset$ is lower or equal than any map. The maps $m_0 = \{(0, 10)\}$ and $m_1 = \{(1, 20)\}$ are not comparable (i.e., $m_0 \not\leq m_1$ and $m_0 \not\geq m_1$) but they have a supremum which is the map $m_2 \stackrel{\text{def}}{=} \{(0, 10), (1, 20)\}$ (this is, $m_0 \leq m_2$, $m_1 \leq m_2$ and also for every other m_2' such that $m_0 \leq m_2'$ and $m_1 \leq m_2'$, then $m_2 \leq m_2'$). On the other hand, the maps $\{(0, 10)\}$ and $\{(0, 20)\}$ are not comparable and they do not have a supremum.

By observing *lock* events the monitor can dismiss some values that are not lower than the verdict, and by observing *free* events the monitor can dismiss some values that are not greater than the verdict. We will see why in Lemma 8.

We classify the property of resource ownership using gray box monitoring and with respect to system assumptions. We first define two possible assumptions about the system:

$$
\begin{aligned}
\mathit{willFree} &\stackrel{\text{def}}{=} \Box((p, r, \mathit{lock}) \rightarrow \Diamond(p, r, \mathit{free})) \\
\mathit{willLock} &\stackrel{\text{def}}{=} \Box(r \notin \mathit{keys}(\mathit{acquired}) \rightarrow \Diamond(_, r, \mathit{lock}))
\end{aligned}
$$

The assumption *willFree* limits the traces to those in which whenever a process locks a resource, then at some point in the future the process will free the resource. Similarly, *willLock* restricts to traces in which whenever a resource is available, some process will eventually lock it.

Let us also consider two functions that override events.

$$
\mathit{noLock}(e) \stackrel{\text{def}}{=} \begin{cases} \mathit{nop} & \text{if } e = (_, _, \mathit{lock}) \\ e & \text{otherwise.} \end{cases} \qquad \mathit{noFree}(e) \stackrel{\text{def}}{=} \begin{cases} \mathit{nop} & \text{if } e = (_, _, \mathit{free}) \\ e & \text{otherwise.} \end{cases}
$$

The function *noLock* overrides *lock* events with *nop* events, while the function *noFree* overrides *free* events with *nop* events.

We can use the functions *noLock* and *noFree* in the observability abstraction *obs* to represent the inability of the monitor to perceive *lock* or *free* events. We define three functions that abstract behavior to observations using *mapS* to map a function over a set of observations and *mapL* to map a function over a finite sequence of events:

$$
\begin{aligned}
obs_{noLock}(s) &\stackrel{\text{def}}{=} \mathit{mapS}\ (\mathit{mapL}\ \mathit{noLock})\ (obs_{\sqsubset}(s)) \\
obs_{noFree}(s) &\stackrel{\text{def}}{=} \mathit{mapS}\ (\mathit{mapL}\ \mathit{noFree})\ (obs_{\sqsubset}(s)) \\
obs_{blind}(s) &\stackrel{\text{def}}{=} \mathit{mapS}\ (\mathit{mapL}\ (\mathit{noLock} \circ \mathit{noFree}))\ (obs_{\sqsubset}(s))
\end{aligned}
$$

An observation of a behavior using obs_{noLock} is a finite subsequence of the behavior where all the *lock* events have been replaced by *nop*. Analogously, obs_{noFree} represents the inability of the monitor to perceive *free* events, and obs_{blind} represents the inability of the monitor to perceive both *lock* and *free* events.

Lemma 8. *The property acquired can belong to any class depending on the assumptions of the system and on the events the monitor can actually perceive.*

Proof. We sketch here the proof that *acquired* is a **Guarantee** property under the assumption *willFree* with observation function $obs_{\sqsubset}$. Let m be the map at the beginning of the monitoring, and let $m' \not\supseteq m$. This means that there is a $(r, p) \in m$ which is not in m'. In other words, there is a resource r which has been acquired by a process p and has not yet been freed. Due to the assumption, the process p will eventually free r, conveying the information that (r, p) was part of the initial map and at that point the monitor can dismiss m' as a candidate.

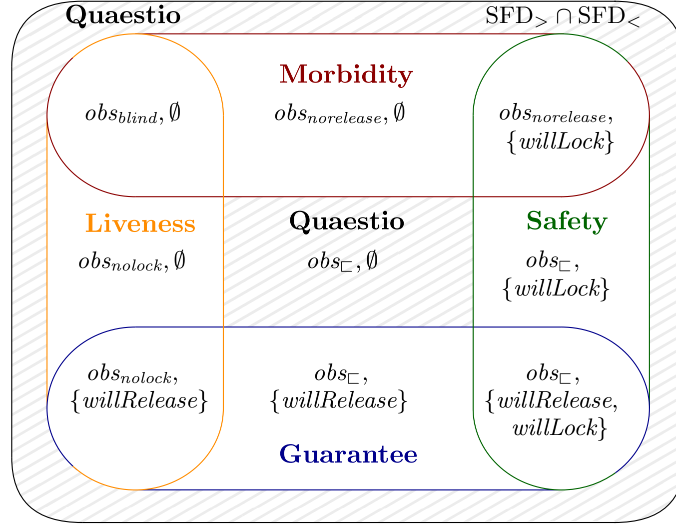

Fig. 3. Classifications of *acquired* with respect to observability and assumptions

We will see that the property is $\mathrm{SFD}_{\not\leq}$ under the assumption *willFree*. Let m be the map at the beginning of the monitoring, and let $m' \not\leq m$. This means that there is a $(r, p) \in m'$ which is not in m. If the monitor witnesses the lock of r, it can dismiss m', but it cannot dismiss maps $m' \not\leq m$ that contain as keys resources that are not locked after the monitoring starts. $\square$

We explain now the classification of the property *acquired* based on value-dismissibilty with respect to *obs* and system assumptions (see Fig. 3).

– *acquired* is a **Guarantee** property under the assumption *willFree*. Let m be the map at the beginning of the monitoring, and let $m' \not\geq m$. This means that there is a $(r, p) \in m$ which is not in m'. In other words, there is a resource r which has been acquired by a process p and not yet released. Due to the assumption, the process p will eventually release r, conveying the information that (r, p) was part of the initial map and at that point the monitor can dismiss m' as a candidate. The property is $\mathrm{SFD}_{\not\leq}$ under the assumption *willFree*: let m be the map at the beginning of the monitoring, and let $m' \not\leq m$. This means that there is a $(r, p) \in m'$ which is not in m. If the monitor witnesses the lock of r, it can dismiss m', but it cannot dismiss maps $m' \not\leq m$ that contain as keys resources that are not locked after the monitoring starts.

– *acquired* is a **Safety** property under the assumption *willLock*. The proof is analogous to the previous one. Let m be the map at the beginning of the monitoring, and let $m' \not\leq m$. This means that there is a $(r, p) \in m'$ which is not in m. Due to the assumption, some process will eventually lock r, conveying the information that (r, p) was not part of the initial map and at that point the monitor can dismiss m' as a candidate. We will see that the property is $\mathrm{SFD}_{\not\geq}$ under the assumption *willLock*. Let m be the map at the beginning of the monitoring, and

let $m' \not\geq m$. This means that there is a resource r locked in m which is not so in m'. If the monitor witnesses the release of r it can dismiss m', but it cannot dismiss maps $m' \not\leq m$ that contain as keys resources that were locked before monitoring started and are not released after that.

– *acquired* is in both a **Safety** and **Guarantee** property if both assumptions hold, but it does not belong to any of the classes if there are no assumptions regarding lock behavior. The reasoning follows from previous classifications.

– If the observability function is obs_{noFree} and thus the monitor cannot detect *free* events, it cannot dismiss values not greater or equal than the result and belongs to the class **Morbidity**. Let m be the map at the beginning of the monitoring, and let $m' \not\geq m$. This means that there is a resource r locked in m which is not so in m'. Since the monitor cannot witness the release of r, it cannot dismiss m'. The property is $\text{SFD}_{\not\leq}$: let m be the map at the beginning of the monitoring, and let $m' \not\leq m$. There exists an $(r, p) \in m'$ which is not in m. If the monitor witnesses the lock of r it can dismiss m', but it cannot dismiss maps $m' \not\leq m$ with resources that are not locked after the monitoring starts.

– If the observability function is obs_{noLock} the property belongs to the class **Liveness**. Let m be the map at the beginning of the monitoring, and let $m' \not\leq m$. This means that there is a $(r, p) \in m'$ which is not in m. Since the monitor cannot witness the lock of r, it cannot dismiss m'. The property is $\text{SFD}_{\not\geq}$: let m be the map at the beginning of the monitoring, and let $m' \not\geq m$. This means that there is a resource r locked in m which is not so in m'. If the monitor witnesses the release of r it can dismiss m', but it cannot dismiss maps $m' \not\leq m$ with resources that were locked before monitoring started and are not released after that.

– If the observability function is obs_{blind} and thus the monitor can detect neither *lock* nor *free* events, it cannot dismiss any map and belongs to both **Morbidity** and **Liveness**. This follows from the reasoning of the previous items.

– *acquired* is a **Liveness** and **Guarantee** property under the assumption *willFree* with the observability function obs_{noLock} because obs_{noLock} makes the property a **Liveness** property, and the assumption makes it a **Guarantee** property.

– *acquired* is **Safety** and **Morbidity** under the assumption *willLock* if the observability function is obs_{noFree} because obs_{noFree} makes the property a **Morbidity** property, and the assumption makes it a **Safety** property.

6 Conclusion

In this paper we have presented a generalization of the classification of Havelund and Peled [20] to expressive verdicts. We have introduced general definitions for admissibility and dismissibility of verdicts and instantianted these to totally ordered and partially ordered domains. Then we have illustrated the taxonomy to quantitative logics like quantitative LTL and discounting LTL. Future work includes studying other quantitative logics like *Counting LTL* [24], where the semantics distinguish the steps necessary until satisfactions, and *Robust LTL* [35]. We also plan to extend our framework to general verdicts in the setting of stream runtime verification.

References

1. Shaull Almagor, Udi Boker, and Orna Kupferman. Discounting in LTL. In *Proc. of the 20th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 424–439. Springer, 2014.
2. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.
3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. of the 5th Int'l Conf on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
4. Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
5. David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.
6. Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engingeering and Methodology*, 20(4):14, 2011.
7. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly—but how ugly is ugly? In *Proc. of the 7th Int'l Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.
8. Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In W. Kuich, editor, *Automata, Languages and Programming*, pages 474–486, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
9. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Proc. of the 13th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 135–149. Springer, 2008.
10. Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Proc. of the 21st. Brazilian Symp. on Formal Methods (SBMF'18)*, volume 11254 of *LNCS*. Springer, 2018.
11. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Proc. of the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE CS Press, 2005.
12. Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proc. of the 23rd Int'l Joint Conf. on Artificial Intelligence (IJCAI'14)*, pages 854—-860. AAAI Press, 2013.
13. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of the 15th Int'l Conf on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
14. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*, volume 85 of *LNCS*, pages 169–181. Springer, 1980.
15. Marco Faella, Axel Legay, and Mariëlle Stoelinga. Model checking quantitative linear time logic. *Electronic Notes in Theoretical Computer Science*, 220(3):61–

77, 2008. Proc. of the Sixth Workshop on Quantitative Aspects of Programming Languages (QAPL 2008).
16. Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.
17. Felipe Gorostiaga, Luis Miguel Danielsson, and César Sánchez. Unifying the time-event spectrum for stream runtime verification. In *Proc. of 20th Int'l Conf. on Runtime Verification (RV'20)*, volume 12399 of *LNCS*, pages 462–481. Springer, 2020.
18. Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 282–298. Springer, 2018.
19. Klaus Havelund and Allen Goldberg. Verify your runs. In *Proc. of the First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05)*, volume 4171 of *LNCS*, pages 374–383. Springer, 2005.
20. Klaus Havelund and Doron Peled. Runtime verification: From propositional to first-order temporal logic. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 90–112. Springer, 2018.
21. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proc. of the 8th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
22. Thomas A. Henzinger and N. Ege Saraç. Monitorability under assumptions. In *Proc. of the 20th Int'l Conf. on Runtime Verification (RV'20)*, volume 12399 of *LNCS*, pages 3–18. Springer, 2020.
23. Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. What can we monitor over unreliable channels? *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2020.
24. Francois Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting LTL. In *Proc. of the 2010 17th Int'l Symp. on Temporal Representation and Reasoning (TIME'10)*, pages 51–58. IEEE, 2010.
25. Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. In *Proc. of the 19th Int'l Conf. on Runtime Verification (RV'19)*, volume 11757 of *LNCS*, pages 273–291. Springer, 2019.
26. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
27. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proc. of the 14th Int'l Symp. on Formal Methods (FM'06)*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.
28. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.
29. Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. of the 20th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 357–372. Springer, 2014.
30. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

31. César Sánchez. Online and offline stream runtime verification of synchronous systems. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 138–163. Springer, 2018.
32. Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.
33. Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Gray-box monitoring of hyperproperties. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019*, volume 11800 of *Lecture Notes in Computer Science*, pages 406–424. Springer, 2019.
34. Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonarkdarpour. Gray-box monitoring of hyperproperties with an application to privacy. *Formal Methods in Systems Desing*, pages 1–36, 2020.
35. Paulo Tabuada and Daniel Neider. Robust linear temporal logic. In *Proc. of the 25th EACSL Annual Conference on Computer Science Logic (CSL'16)*, volume 62 of *LIPIcs*, pages 10:1–10:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
36. Xian Zhang, Martin Leucker, and Wei Dong. Runtime verification with predictive semantics. In *Proc. of the 4th Int'l Symp NASA Formal Methods (NFM'12)*, LNCS, pages 418–432. Springer, 2012.