On Monitoring Linear Temporal Properties

Klaus Havelund[1] and Doron Peled[2]

[1]Laboratory for Reliable Software, Jet Propulsion Laboratory, Pasadena, 91109, California, USA.
[2]Department of Computer Science, Bar Ilan University, Ramat Gan, 5290002, Israel.

Contributing authors: klaus.havelund@jpl.nasa.gov; doron.peled@gmail.com;

Abstract

Runtime verification facilitates monitoring the executions of a system against temporal properties, commonly to detect violations. Not every temporal property is fully monitorable however: in some cases, a positive or negative verdict on the monitored execution does not depend on any finite prefix of it. We study the problem of monitoring properties written in Linear Temporal logic (LTL). We provide a complete classification of the temporal properties based on the ability to provide positive and/or negative verdicts in finite time.

Keywords: Runtime verification, monitorability, property classification, linear temporal logic

1 Introduction

Model Checking [8, 12, 32] provides algorithms and methods for the exhaustive verification of (models of) finite state systems against their formal specification. It has also been extended to deal, to some limited extent, with infinite state systems, e.g., for a single stack machine [7]. Runtime verification (RV) facilitates the direct monitoring of the execution of a system, checking it against a formal specification. This can be useful for, e.g., testing a system before it is deployed, to reduce potential errors, as well as monitoring the system after its deployment in order to detect or avert failures. RV can be applied directly to a monitored execution, possibly as it happens, and it is not limited to monitoring finite state systems. On the other hand, a verdict, positive

(whether the monitored property holds) or negative (whether it fails to hold) needs to be given after inspecting a finite prefix of the execution. While the complexity of RV is quite reasonable, in comparison with model checking, sampling executions with RV techniques can only increase the belief of reliability of the monitored system: it is not an exhaustive check, rather, one execution is checked at a time, hence it does not provide the same level of guarantee as model checking.

RV can be applied to improve the reliability of safety critical and mission critical systems, including safety as well as security aspects, and can more generally be applied for processing streaming information. Often, the stream of information is not a priori limited to a specific length, and the monitored property is supposed to follow the execution for as long as it is running. It is essential to keep the *incremental time complexity*, required to update the monitoring algorithm between successively observed events, as small as possible, in order to be able to follow the speed of the monitored execution.

The RV specification properties, against which the system is monitored, are often given in *linear temporal logic* (LTL) [27]. These properties are traditionally interpreted over infinite execution sequences (the monitored system keeps emitting events). But for runtime verification to be useful, it is necessary to be able to provide a verdict after observing a finite prefix of an execution sequence, (also referred to as just a *prefix*). For example, the property $\Box p$ (for some atomic proposition p), which asserts that p always holds throughout the execution, can be *refuted* by a runtime monitor, i.e., demonstrating a negative verdict, if p does not hold in some observed event. At this point, no matter which way the execution is extended, the property will not hold. However, no finite prefix of an execution can guarantee a positive verdict that $\Box p$ holds, since no matter how long we have observed that p has been holding, it may still stop holding in some future. In a similar way, the property $\Diamond p$ cannot be refuted, since p may hold at any time in the future; on the other hand, once p holds, we already establish that the property is satisfied, independent on any continuation, and we can issue a positive verdict. For the property $(\Box p \vee \Diamond q)$ we may not have a verdict at any finite time when monitoring the execution where all the observed events satisfy both p and $\neg q$. On the other hand, we may never "lose hope" to have such a verdict, as a later state satisfying q will result in a positive verdict. For the property $\Box\Diamond p$ we can never provide a verdict in finite time: for whatever happens, even if p holds an infinite number of times, we cannot *guarantee* or refute that this property holds when observing any finite prefix of an execution. The *monitorability problem* of a temporal property was studied in [5, 14, 31]. There, a property is considered to be monitorable if after monitoring any finite prefix, we still have a possibility to obtain a positive or a negative verdict in a finite number of steps.

We refine here the study of LTL monitorability by distinguishing cases where *some* verdict is always possible, *no* verdict is ever possible, or some verdict is possible now, but no verdict may be possible later, depending on the monitored prefix. We extend Lamport's *safety* and *liveness* classification of temporal properties with *guarantee*, which is defined to be the dual of *safety* in [27], i.e., the negation of a safety property is a guarantee property and vice versa, and *morbidity*, which we define as the

dual of *liveness*. To complete this classification to cover all possible temporal specifications, we add another class, which we term *quaestio*. We study the relationship between this classification and monitorability. In particular, the *safety* class includes the properties whose failure can be detected after a finite prefix, and the *liveness* properties are those where one can never conclude a failure after a finite prefix.

We suggest some variants for runtime verification algorithms that take the refined notions of monitorability into account before and during runtime verification. Equipped with these algorithms, we can check what kind of verdicts one can expect a priori from monitoring an execution against a given temporal specification, and can also update this expectation during runtime when some verdicts are not possible anymore. In addition, these algorithms can be used to decide whether a given specification is a *safety*, *guarantee*, *liveness*, *morbidity* or *quaestio* property.

Related work. Alpern and Schneider [1] formalized Lamport's definition of *safety* and *liveness*. Sistla [34] showed a PSPACE algorithm for checking *safety*, and an EXPSPACE algorithm for checking *liveness*. Checking *liveness* was shown to be in EXPSPACE-complete in [23]. Drissi-Kaitouni and Jard [11], as well as Kupferman and Vardi [24] studied the problem of monitoring LTL properties for an execution sequence. Pnueli and Zaks [31] proposed constructing compositional testers for runtime verification. They also considered the issue of monitorability of a property, requiring that any finite prefix can be extended in a finite manner such that a positive or negative verdict can be reported in finite time. Finally, they provided a tester based algorithm for checking whether an observed finite prefix can be extended in a finite way to obtain a positive or a negative verdict. Fernandez, Jard, Jéron and Viho supported checking for availability of future verdicts for a given test objective in the TGV test case generator [15]. Bauer, Leucker and Schallhart [5] defined prefixes that cannot be finitely extended to obtain a verdict for a temporal specification as *ugly* prefixes; then they defined a property to be monitorable if it has no *ugly* prefixes. They showed that *safety* and *guarantee* properties are monitorable, but there are some other monitorable properties that are not in these classes. Diekert and Leucker [10] studied monitorability and its connection to safety and *liveness* using topological characterizations. Falcone, Fernandez and Mounier [13] considered the Manna-Pnueli hierarchy of properties and showed that some of the classes of this hierarchy have both monitorable and non-monitorable properties.

Contribution. We revisit the classification of properties according to *safety*, *guarantee* and *liveness* after completing it to cover all the temporal properties. We add new classes of properties. The first one we call *morbidity*; it is the dual class to *liveness*, i.e., a negation of a *liveness* property is a *morbidity* property and vice versa. To complete the space of temporal properties, we add another class called *quaestio*.

We provide an alternative definition for these classes that is based on the possible results one can obtain during runtime monitoring; this depends on whether one can always/sometimes/never obtain a positive or a negative verdict based on a finite trace. Then we study a refinement of runtime monitorability with respect to these classes and their intersections.

We propose an assortment of algorithms for runtime verification, which extend the classical LTL runtime verification algorithm. These variants allow us to decide a

priori what kind of verdicts are expected from a property, and also update the possibilities as the monitored execution unfolds. Because of the close connection between the discussed classification and notions of monitorability, they can also be used to identify the class of a given LTL specification.

This paper is an extended version of the preliminary paper version in [29]. We show the relation of the classification presented here with the Manna-Pnueli hierarchy. We provide more details about the classification, analysing how prefixes can be extended to move between classes of the hierarchy.

Overview of paper.

The paper is organized as follows. Section 2 provides some preliminary introductions to selected concepts, including runtime verification, linear temporal logic and monitorability. Section 3 presents our refinement of Lamport's classification of temporal properties, associated with the concept of monitorability. Section 4 introduces algorithms for determining monitorability and classification of temporal properties. Section 5 concludes the paper.

2 Preliminaries

2.1 Runtime Verification

Runtime verification (RV) [2, 18] very generally refers to the use of rigorous (formal) techniques for *processing* execution traces emitted by a system being observed. In general, the purpose of RV is to evaluate the state of the observed system. Since only single executions (or collections thereof) are analyzed, RV scales well compared to more comprehensive formal methods, but of course at the cost of coverage. In runtime verification one is not concerned with how to obtain various executions, as in e.g. test case generation. This reflects a focus of attention (research) rather than a judgment of utility – test case generation is of critical importance.

An execution trace is generated by the observed executing system, typically by instrumenting the system to generate events when important transitions take place. Instrumentation can be manual by inserting logging statements in the code, or it can be automated using instrumentation software, such as e.g. aspect-oriented programming frameworks. In the extreme case, an event can represent a complete view of the internal state of the system. Processing can take place on-line, as the system executes, or off-line, by processing log files produced by the system. In the case of on-line processing, observations can be used to control (shield) the monitored system [6].

Processing can take numerous forms. We focus here on *specification-based* runtime verification, where an execution trace is checked against a property expressed in a formal (usually temporal) logic. Expressed more formally, assume an observed system S, and assume further that a finite execution of S up to a certain point is captured as an execution trace $\xi = e_1.e_2.\ \ldots\ .e_n$, which is a sequence of observed events. Assume the type $\mathbb{E}$ of events; then the RV problem can be formulated as constructing a program $M : \mathbb{E}^* \to D$, which when applied to the trace ξ, as in $M(\xi)$, returns some data value $d \in D$ in a domain D of interest. In specification-based RV, typically M is generated from a formal specification, given e.g. as a temporal logic formula, a state

machine, or a regular expression, and d is a *verdict* in the Boolean domain ($d \in \mathbb{B}$), or some extension of the Boolean domain as discussed in [4], indicating whether the execution trace conforms to the specification.

However, RV should be perceived broadly, e.g. d can be a visualization of the execution trace, a learned specification (specification mining), statistical information about the trace, an action to perform on the running system S, etc. The problem can be even further generalized to computing a result from multiple traces, as e.g. done in specification learning [20–22, 30] and statistical model checking [26], giving M the type $M : 2^{\mathbb{E}^*} \to D$.

That execution trace is often unbounded in length, representing the fact that the observed system "keeps running", without a known termination point. Hence it is important that the monitoring program is capable of producing verdicts based on (unbounded) finite prefixes of the execution trace observed so far. The remainder of the paper discusses what kind of verdicts can be produced from finite prefixes given a specific property.

2.2 Linear Temporal Logic

The classical definition of linear temporal logic is based on future modal operators [27] with the following syntax:

$$\varphi ::= \mathit{true} \mid p \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \,\mathcal{U}\, \varphi) \mid \bigcirc\varphi$$

where p is a proposition from a finite set of propositions P, with $\mathcal{U}$ standing for *until*, and $\bigcirc$ standing for *next-time*. One can also write $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\Diamond\varphi = (\mathit{true}\,\mathcal{U}\,\varphi)$ (for *eventually* φ) and $\Box\varphi = \neg\Diamond\neg\varphi$ (for *always* φ).

An event e is a subset of the propositions in P. These are the propositions that *hold* in that event. A *trace* $\sigma = e_0.e_1.e_2\ldots$ is an infinite sequence of events. We denote the event e_i in σ by $\sigma(i)$. LTL formulas are interpreted over an infinite sequence of *events*. LTL semantics is defined as follows:

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

Then $\sigma \models \varphi$ when $\sigma, 0 \models \varphi$.

2.3 Past Propositional Temporal Logic

We continue with a standard definition of past-time propositional linear time temporal logic PLTL. Let P be a finite set of *propositions*. Then the syntax of PLTL is as follows:

$$\varphi ::= \mathit{true} \mid p \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \,\mathcal{S}\, \psi) \mid \ominus\varphi$$

where $p \in P$. We can use the following additional operators: $false = \neg true$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\mathbf{P}\,\varphi = (true\,\mathcal{S}\,\varphi)$, $\mathbf{H}\,\varphi = \neg\mathbf{P}\,\neg\varphi$.

The operator $\ominus$ (for *previous-time*) is the past mirror of the $\bigcirc$ operator. Similarly, **P** (for *Previous*) is the past mirror of $\Diamond$, **H** (for *history*) is the past mirror of $\Box$ and $\mathcal{S}$ (for *Since*) is the past mirror of $\mathcal{U}$.

A *trace* $\sigma = e_0.e_1 \ldots e_n$ is a finite sequence of events, consisting each of a subset of the propositions P. We denote the length of the sequence of events $\sigma = e_1.e_2 \ldots e_n$ as $|\sigma| = n$.

Semantics. The semantics of a PLTL formula φ with respect to a finite trace σ is defined as follows:

- $\sigma, i \models true$.
- $\sigma, i \models p$ iff $p \in \sigma(i)$.
- $\sigma, i \models (\varphi \wedge \psi)$ iff $\sigma, i \models \varphi$ and $\sigma, i \models \psi$.
- $\sigma, i \models \neg\varphi$ iff not $\sigma, i \models \varphi$.
- $\sigma, i \models \ominus\varphi$ iff $|\sigma| > 1$ and $\sigma, i-1 \models \varphi$.
- $\sigma, i \models (\varphi\,\mathcal{S}\,\psi)$ iff for some $j \leq i$, $\sigma, j \models \psi$, and for each k such that $j < k \leq i$, $\sigma, k \models \varphi$.

Then, for a finite sequence σ with length $|\sigma| = n$, we define $\sigma \models \varphi$ iff $\sigma, n \models \varphi$. We define four extensions of PLTL, which are prefixed with one or two future operators from $\{\Diamond, \Box\}$ and may further contain only past operators. All extensions are interpreted over infinite sequences:

- $\Box$LTL, which consists of PLTL formulas prefixed with the future $\Box$ operator.
- $\Diamond$LTL, which is, similarly, PLTL formulas prefixed by the $\Diamond$ operator.
- $\Box\Diamond$LTL, which consists of past formulas prefixed with $\Box\Diamond$.
- $\Diamond\Box$LTL, which consists of past formulas prefixed with $\Diamond\Box$.

Note the duality between the first two restricted versions of LTL, $\Box$LTL and $\Diamond$LTL: for every formula φ, $\neg\Box\varphi = \Diamond\neg\varphi$. Thus, the negation of a $\Box$LTL property is a $\Diamond$LTL property. Similarly, for every φ, $\neg\Box\Diamond\varphi = \Diamond\Box\neg\varphi$, making the latter two restricted versions of LTL also dual. Thus, the negation of a $\Box\Diamond$LTL property is a $\Diamond\Box$LTL property.

2.4 Monitorability

Bauer, Leucker and Schallhart [5] define three categories of observed sequences of events over 2^P.

- A *good* prefix is one where all its extensions (with infinite sequences of elements from 2^P) satisfy the monitored property φ.
- A *bad* prefix is one where none of its infinite extensions satisfies φ.
- An *ugly* prefix cannot be extended into a *good* or a *bad* prefix.

When identifying a *good* or a *bad* finite prefix, we can stop tracing the execution and can announce that the monitored property is *satisfied* or *failed*, respectively. After an *ugly* prefix, satisfaction or refutation of φ depends on the entire infinite execution, and cannot be determined in finite time.

Monitorability of a property φ is defined in [5] as the lack of *ugly* prefixes for the property φ. This definition is consistent with an early definition in [31].

Ugly prefixes cannot occur in an execution satisfying a *safety* property [5]. To see this, observe that an *ugly* prefix σ cannot be extended into a *good* or *bad* prefix, hence it must have both an infinite extension that satisfies the property, and, in particular, another one ρ that does not satisfy it. But then there is no prefix of ρ that is *bad*, otherwise σ would not be ugly. But this contradicts the definition of a safety property.

Thus, every *safety* property is monitorable. Because *guarantee* properties are the negations of *safety* properties, one obtains using a symmetric argument that every *guarantee* property is also monitorable.

3 Characterizing Temporal Properties According to Monitorability

Safety and *liveness* temporal properties were defined informally on infinite execution sequences by Lamport [25] as *something bad cannot happen* and *something* good *will happen*. These informal definitions were later formalized by Alpern and Schneider [1]. *Guarantee* properties where used in an orthogonal characterization by Manna and Pnueli [27]; *guarantee* properties are the dual of *safety* properties, that is, the negation of a *safety* property is a *guarantee* property and vice versa. We add to this picture *morbidity* properties, which is the dual class of *liveness* properties.

- *safety*: A property φ is a *safety* property, if for every execution that does not satisfy it, there is a finite prefix such that completing it in any possible way into an infinite sequence would not satisfy φ.
- *guarantee* (co-*safety*): A property φ is a *guarantee* property, if for every execution satisfying it, there is a finite prefix such that completing it in any possible way into an infinite sequence satisfies φ.
- *liveness*: A property φ is a *liveness* property if every finite sequence of events can be extended into an execution that satisfies φ.
- *morbidity* (co-*liveness*): A property φ is a *morbidity* property if every finite sequence of events can be extended to an execution that violates φ.

By definition, *safety* and *guarantee* properties, corresponding in LTL to the classes □LTL and ◇LTL defined in 2.3, are dual. That is, the negation of a *safety* property is a *guarantee* property, and vice versa. Similarly, *liveness* and *morbidity* are dual.
Online runtime verification of LTL properties inspects finite prefixes of the execution. Hence, it may sometimes provide only a partial verdict on the satisfaction and violation of the inspected property [4, 28]. This motivates providing three kinds of verdicts:

- *refuted* (or *failed* or *negative*) when the current prefix cannot be extended in any way into an execution that satisfies the specification,
- *established* (or *satisfied* or *positive*) when any possible extension of the current prefix satisfies the specification, and
- *undecided* when the current prefix can be extended to satisfy the specification but also extended to satisfy its negation.

Tracing a *safety* property, there exists always a *bad* (hence, finitely traceable) prefix if it fails to hold in an execution. Correspondingly, there exists always a *good* (hence, again, finitely traceable) prefix when an execution satisfies a *guarantee* property.

Each temporal property is a conjunction of a *liveness* and a safety property, as shown by Alpern and Lamport in [1]. Due to the duality between *safety* and *guarantee* (a negation of a *safety* property is a *guarantee* property, and vice versa) and between *liveness* and *morbidity* (a negation of a *liveness* property is a *morbidity* property and vice versa), we immediately obtain, through De-Morgan Laws that every temporal property is a disjunction of a *guarantee* and a *morbidity* property.

Safety, *guarantee*, *liveness* and *morbidity* can be seen as characterizing finite monitorability of temporal properties: if a *safety* property is violated, there will be a finite *bad* prefix witnessing it; on the other hand, for a *liveness* property, one can never provide such a finite negative evidence. We suggest the following alternative definitions of classes of temporal properties. The adverbs *always* and *never* in the definitions of the classes below correspond to *for all the executions* and *for none of the executions*, correspondingly.

- AFR (*safety*): Always Finitely Refutable: for each execution where the property does not hold, refutation can be identified after a finite (*bad*) prefix, which cannot be extended to an (infinite) execution that satisfies the property.
- AFS (*guarantee*): Always Finitely Satisfiable: For each execution where the property is satisfied, satisfaction can be identified after a finite (*good*) prefix, where each extension of it will satisfy the property.
- NFR (*liveness*): Never Finitely Refutable: For no execution, can a *bad* prefix be identified after a finite prefix. That is, every finite prefix can be extended into an (infinite) execution that satisfies the property.
- NFS (*morbidity*): Never Finitely Satisfiable: For no execution can a *good* prefix be identified after a finite prefix. That is, every finite prefix can be extended into an (infinite) execution that does not satisfy the property.

It is easy to see that the definitions of the classes AFR and *safety* are the same and so are those for AFS and *guarantee*. We will show the correspondence between NFR and *liveness*. A *liveness* property φ is defined to satisfy that any finite prefix can be extended to an execution that satisfies φ. The definition of the class NFR only mentions prefixes of executions that do not satisfy φ; but for prefixes of executions that satisfy φ this trivially holds. The correspondence between NFS and *morbidity* is shown in a symmetric way.

The above four classes of properties, however, do not cover the entire set of possible temporal properties, independent of the actual formalism that is used to express them. The following two classes complete the classification.

- SFR: Sometimes Finitely Refutable: for some infinite executions that violate the property, refutation can be identified after a finite (*bad*) prefix; for other infinite executions violating the property, this is not the case.
- SFS: Sometimes Finitely Satisfiable: for some infinite executions that satisfy the property, satisfaction can be identified after a finite (*good*) prefix; for other infinite executions satisfying the property, this is not the case.

Let φ be any property expressible in LTL. Then φ represents the set of executions satisfying it. It is clear by definition that φ must be either in AFR, SFR or in NFR (since this covers all possibilities). It also holds that φ must be in either AFS, SFS or in NFS. Every temporal property must belong then to a class XFR, where X stands for A, S or N, and also to a class XFS, again with X is A, S or N. We call it the FR/FS classification. The FR/FS classification refines the classification of properties as *safety*, *guarantee*, *liveness* and *morbidity*, in the sense of further dividing these into subclasses as shown in Figure 1. Specifically, it identifies the intersections between these classes. Below we give examples for the nine combinations of XFR and XFS, appearing in clockwise order according to Figure 1.

- SFR ∩ NFS: $(\Diamond p \wedge \Box q)$
- AFR ∩ NFS: $\Box p$
- AFR ∩ SFS: $(p \vee \Box q)$
- AFR ∩ AFS: $\bigcirc p$
- SFR ∩ AFS: $(p \wedge \Diamond q)$
- NFR ∩ AFS: $\Diamond p$
- NFR ∩ SFS: $(\Box p \vee \Diamond q)$
- NFR ∩ NFS: $\Box\Diamond p$
- SFR ∩ SFS: $((p \vee \Box\Diamond p) \wedge \bigcirc q)$

These nine possibilities are pairwise disjoint.

The set of all properties *Prop* is not covered by *safety*, *guarantee*, *liveness* and morbidity. The missing properties are in SFR ∩ SFS. We call this latter class of properties *Quaestio* (Latin for *question*).

Observe that for AFR ∩ AFS we gave an example of a property with only the nexttime operator ○. We show that for LTL, any property φ in AFR ∩ AFS can be written with only the nexttime and the Boolean operators. To see this, consider a tree whose edges are labeled with elements from 2^P; every finite path from the root down is labeled with a prefix of a *minimal good* prefix[1] for φ. That is, if a prefix is *good* then the path terminates in a leaf node. This is a finitely branching tree, since the number of successors of each node are at most $2^{|P|}$. Assume that this tree has an infinite path. This path must satisfy φ, as, being a *safety* property, if this path does not satisfy φ, it has a *bad* prefix, which cannot be extended to satisfy φ. But φ is also a *guarantee* property, hence it must have a finite *good* prefix. But according to the construction, a *good* prefix leads to a leaf node and is not extended in the tree, contradicting the assumption that the tree has an infinite path. Since the tree is finite, it is easy to see that one can express φ in LTL based on the finitely many *good* prefixes (paths) in the tree using ○ and the Boolean operators[2]. The converse also holds: any property that is expressible in this way corresponds to such a finite tree, and thus is in the intersection of the classes *safety* and *guarantee*.

[1] A finite extension of a *good* (*bad* or *ugly*) prefix remains *good* (*bad* or *ugly*, respectively).

[2] One can also use other operators to express the same property, e.g., by adding a trivial disjunct, as in $(\varphi \vee (\Box p \wedge \Diamond \neg p))$, which is equivalent to φ.

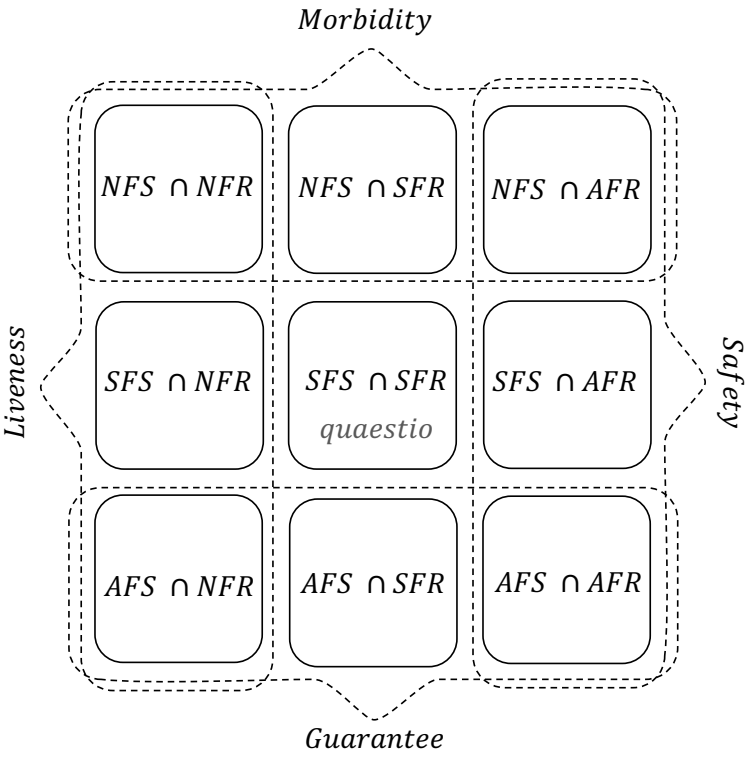

Fig. 1: Classification of properties: *safety*, *guarantee*, *liveness*, *morbidity* and *quaestio*.

The Manna and Pnueli Characterization and Monitorability

Manna and Pnueli [27] presented a different characterization of families of temporal properties, which is orthogonal to Lamport's *safety/liveness* characterization, and its extension presented in this section. They showed a correspondence between the LTL *safety* and the logic □LTL, and between the LTL *guarantee* properties as ◇LTL. They presented a hierarchy of families of properties that can express any LTL property as shown in Figure 2. They also showed a corresponding topological characterization.

The *safety* properties are identified by Manna and Pnueli with the □LTL properties, that is, each (future) LTL property can be written equivalently in □LTL. Similarly, *guarantee* properties can be written as ◇LTL properties. It is easy to see that *guarantee* properties are the complements of *safety* properties. These two classes are the same as the classes with the same names presented earlier in this section. The *obligation* class consists of Boolean combinations of *safety* (and, consequently, *guarantee* properties). Further up the hierarchy are *response* properties, identified by Manna and Pnueli with the syntactic class of properties □◇LTL, and *persistence* with ◇□LTL. These two classes of properties are also negations of each other. Finally, *obligation* properties are Boolean combinations of *response* (and, consequently, also

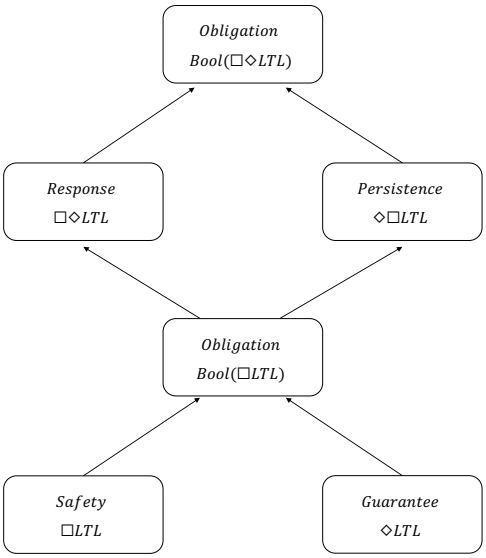

Fig. 2: Classification of properties according to classes of properties.

persistence properties). As we move up the hierarchy (with the arrows in Figure 2) the upper classes include the lower classes.

Bauer, Leucker and Schallhart [5] showed that *safety* and *guarantee* properties are monitorable. Falcone, Fernandez and Mounier [13] showed that *obligation* properties are monitorable. This was done by stating[3] that monitorability is closed under the Boolean operators.

Lemma 1 *Monitorability is closed under the Boolean operators.*

Proof. First, observe that monitorability is closed under negation, since by negating a property *good* and *bad* prefixes are switched with each other and other prefixes remain undecided w.r.t. the monitored property. Now, consider the conjunction $(\varphi \wedge \psi)$ of two monitorable properties φ and ψ. Any *bad* prefix of either φ or ψ is a *bad* for the conjunction. Now, assume a prefix that does not have a *bad* extension for either φ or ψ. Then it must have a *good* prefix for each one of them. Since a *good* prefix can only be extended to a *good* prefix, the longer of the prefixes is good for the conjunction. The argument for the disjunction of properties is symmetric. □

In [13] examples of a non-monitorable *response* property (written in plain LTL as $\Box(r \rightarrow \Diamond q)$) and of a monitorable *response* property (written in plain LTL as $\Box((r \rightarrow \Diamond q) \wedge \neg(r \wedge \bigcirc r))$ are given.

[3] This was done without a proof, hence, for completeness we detail the proof here.

3.1 Refining monitorability

Before any verdict, positive or negative, on the satisfiability of a temporal formula φ is given, the current inspected execution prefix can be extended to satisfy or falsify the property. There are four possibilities into which we can extend a finite sequence:

- fin sat: a *good* prefix, i.e., where all further extensions satisfy the property.
- inf sat: an infinite extension satisfying the property, where none of its prefixes is a good prefix.
- fin ref: a *bad* prefix, i.e., where all further extensions do not satisfy the property.
- inf ref: an infinite extension that does not satisfy the property, where none of its prefixes is a *bad prefix*.

The definitions of the classes NFS, SFS, AFS, NFR, SFR and AFR directly dictates which combination of the above four possibilities are initially available for the different cases. For example, for the class NFR, executions that do not satisfy the property can only be of type inf ref sequences, since no execution can be finitely refutable. For the class SFS, we have both executions that can be finitely satisfiable, and executions that satisfy the property that do not have *good* prefixes. The class NFR∩SFS contains executions of types inf ref, inf sat and fin sat. As the monitoring of an execution progresses, the possibilities to a achieve a positive or a negative verdict may diminish, and similarly the possibility to have an infinite extension that satisfies or falsifies the property. This is indicated in Figure 3 using the arrows. For each one of the nine intersections between classes of properties (NFR, SFR and AFR intersection with NFS, SFS and AFS), we indicate which one of the possibilities using arrows between areas that correspond to different classes of properties. A prefix extended by a *single event* may sometimes progress according to a pair of consecutive arrows at once.

Now, when the only possibilities that remain are by refutation or satisfaction by an infinite sequence, the current sequence is necessarily *ugly*. This makes the properties in NFS∩NFR non-monitorable. However, some properties in the classes NFS∩SFR, SFS∩NFR and SFS∩SFR are also non-monitorable, since after some prefix, refutation or satisfaction depends on the entire infinite execution. This is demonstrated in the following table.

Class	monitorable example	non-monitorable example
SFR ∩ SFS	$((\Diamond r \vee \Box\Diamond p) \wedge \bigcirc q)$	$((p \vee \Box\Diamond p) \wedge \bigcirc q)$
SFR ∩ NFS	$(\Diamond p \wedge \Box q)$	$(\Box\Diamond p \wedge \bigcirc q)$
NFR ∩ SFS	$(\Box p \vee \Diamond q)$	$((\neg p \,\mathcal{U}\, \Diamond(p \wedge \bigcirc\neg p)) \vee \Box\Diamond p)$

Consider for example the property $((p \vee \Box\Diamond p) \wedge \bigcirc q)$, which is in SFR ∩ SFS. If p holds in the first event, then there are the following extensions: a *good* one, where q holds in the second event, and a *bad* one if it does not hold. On the other hand, if p does not hold in the first event, then there is a possibility of extending the situation into a *bad* prefix, if q does not hold in the second event. Otherwise, we obtain an *ugly* prefix, since no good or *bad* extension is possible anymore.

We propose that RV can still be applied for non-monitorable properties if at least initially some verdicts can be made. We refine the definition of monitorability into the following categories as follows. Correspondingly, in Figure 4, the dark areas correspond to the non-monitorable properties.

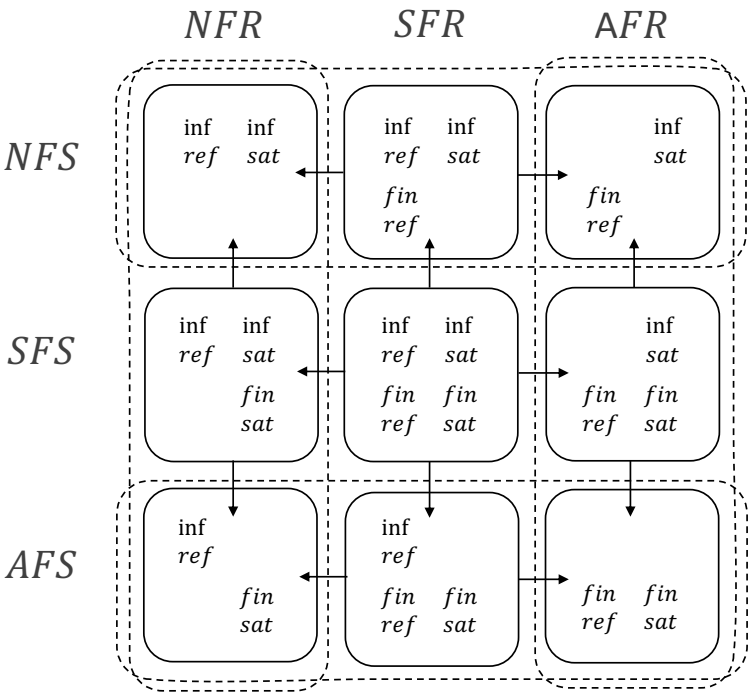

Fig. 3: Classification of properties according to classes of properties: arrows represent how a prefix can evolve from one class to another when extended.

- A property is *monitorable* if it cannot have an *ugly* prefix. This corresponds to the definition of monitorability in [5, 31]. *Safety* and *guarantee* properties are universally monitorable. But as demonstrated above, some of the properties in SFR ∩ SFS, SFR ∩ NFS and NFR ∩ SFS are also monitorable.
  Checking monitorability can be done using Algorithm 3 in Section 4.3.
- A property has *zero monitoring information* if there is no information that can be obtained by monitoring it any finite amount of time. The properties in the intersection of *liveness* and *morbidity* are those that have zero monitoring information.
  Checking that a property has zero monitoring information can be done by applying algorithm 3 (or Algorithm 4 for checking that the property is both in NFR and in NFS).
- A property is *weakly monitorable* if there exist *ugly* prefixes, but not all the finite prefixes are *ugly*. In this case, there is still information that we can obtain by monitoring it, but at times, we may observe an *ugly* prefix, from which no interesting information can be concluded in finite amount of time.
  Algorithm 3 in Section 4 can be used to check that a property is non-monitorable, yet also not in zero monitoring information. In this case, instead of using Algorithm 1 for performing the runtime verification, one can use Algorithm 2 to also

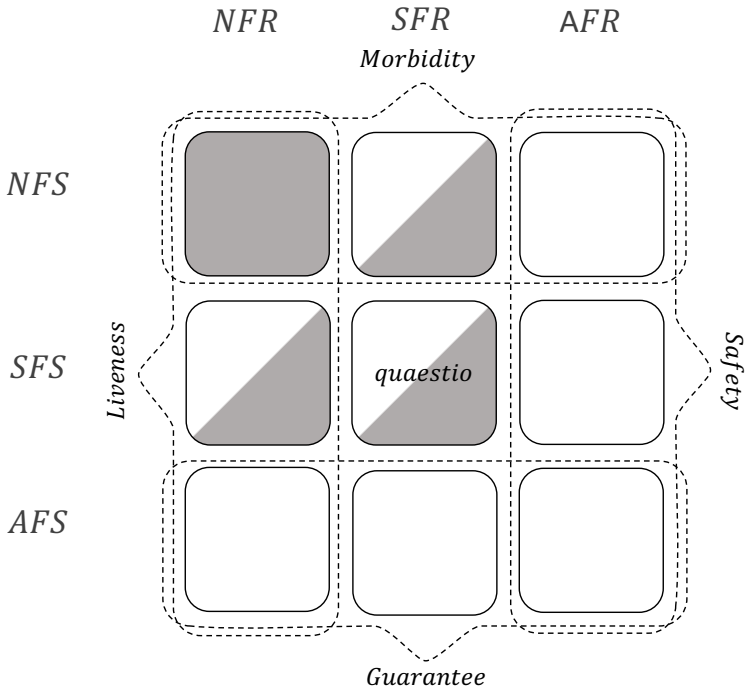

Fig. 4: Classification of properties according to monitorability: filled space correspond to nonmonitorable properties.

check whether *some* verdict is still possible for the current prefix, abandoning the runtime verification when this is not the case. The semi-filled areas in Figure 4 represent the weakly monitorable properties.

Consider the property $(p \vee (\neg q \,\mathcal{U}\, (p \wedge \Box\Diamond r)))$. This property is in SFS $\cap$ SFR, i.e., *quaestio*. It is non-monitorable, as demonstrated by the *ugly* prefix $\{\}.\{p\}$ (i.e., all the propositions are false in the first event, and only p is true in the second event), after which no verdict can be given. We consider it to be weakly monitorable. A priori, we can expect both a positive or a negative verdict: if p holds in the first event, then a positive verdict is given; if q holds before p, then a negative verdict is given.

4 Runtime Verification Algorithms for Monitorability

We present four algorithms related to monitorability of LTL propositions.

1. A description of the classical algorithm for runtime monitoring of LTL (or Büchi automata) properties [24].
2. An algorithm for check during runtime what kind of verdicts can still be produced given the current prefix.
3. An algorithm for checking whether the property is monitorable.

4. An algorithm for checking the class of a given temporal property under the characterization given in this paper.

4.1 Algorithm 1: Monitoring sequences using automata

Kupferman and Vardi [24] presented an algorithm for monitoring execution sequences while providing a *success* (positive) or *fail* (negative) verdict of the checked property, whenever a *good* or a *bad* prefix has already occurred, respectively.

For detecting *good* prefixes, we do the following:

1. Construct a Büchi automaton $\mathcal{A}_{\neg\varphi}$ for $\neg\varphi$, e.g., using the translation in [17]. This automaton is not necessarily deterministic [36].
2. Using DFS, find the states of $\mathcal{A}_{\neg\varphi}$, from which one cannot reach a cycle that contains an accepting state. This can be done by first removing, using Depth First Search (DFS) the states that are unreachable from the initial states. Then, from each of the remaining *accepting* states s, check using DFS whether a cycle through s is possible.
3. Checking for a *positive* (*good*) verdict for φ, one maintains for each monitored prefix the set of states that the automaton $\mathcal{A}_{\neg\varphi}$ will reach after observing that input as follows:
   - One starts with the set of initial states of the automaton $\mathcal{A}_{\neg\varphi}$.
   - Given the current set of successors S and a newly occurring event $e \in 2^P$ that extends the monitored prefix, the next set of successors S' is set to the successors of the states in S according to the transition relation Δ of $\mathcal{A}_{\neg\varphi}$. That is, $S' = \{s' \mid s \in S \wedge (s, e, s') \in \Delta\}$.
   - Reaching the empty set of states, the monitored sequence is *good*, and the property must hold since the current prefix cannot be completed into an infinite execution satisfying $\neg\varphi$.

This is basically a *subset construction* and indeed we can construct a deterministic automaton $\mathcal{B}_\varphi$ as follows.

- The initial state consists of the set initial states of $\mathcal{A}_{\neg\varphi}$ that were not removed.
- The accepting state is the empty set of states.
- The transition relation is as described above.

Translating the formula $\neg\varphi$ into a Büchi automaton can result in an automaton $\mathcal{A}_{\neg\varphi}$ of size $O(2^{|\varphi|})$. The size of the automaton $\mathcal{B}_\varphi$ is $O(2^{2^{|\varphi|}})$, resulting in a double exponential explosion from the size of the LTL property φ. But in fact, we do not need to construct the entire automaton $\mathcal{B}_\varphi$ a priori, and can avoid the double exponential explosion by calculating its current state (which is a subset of the states of $\mathcal{A}_{\neg\varphi}$) on-the-fly, and update it with each incoming event. A positive verdict is given when we reach the empty state. The size of a state of $\mathcal{B}_\varphi$ is exponential in the size of φ, thus, this is also the incremental complexity for processing each monitored event. A single exponential explosion is also a lower bound [24], as shown below.

Checking for a *negative* (*bad*) verdict for φ is done using a symmetric construction, first translating φ into a Büchi automaton $\mathcal{A}_\varphi$ and then the deterministic

automaton $\mathcal{B}_{\neg\varphi}$ (or calculating its states on-the-fly) using a subset construction symmetric to the above. Note that $\mathcal{A}_{\neg\varphi}$ is used to construct $\mathcal{B}_{\varphi}$ and $\mathcal{A}_{\varphi}$ is used to construct $\mathcal{B}_{\neg\varphi}$. Runtime verification of φ uses both automata for the monitored input, reporting a *negative* verdict if $\mathcal{B}_{\neg\varphi}$ reaches an accepting state, a *positive* verdict if $\mathcal{B}_{\varphi}$ reaches an accepting state, and an *undecided* verdict otherwise. The algorithm guarantees to report a *positive* or *negative* verdict on the *minimal good* or *bad* prefix that is observed.

A Lower Bound Example for LTL Monitoring

To complete the picture of the monitorability for LTL, we present the following example, used by Kupferman and Vardi [24], to show that an automaton that is used to monitor an LTL specification may result in a number of states that is doubly exponential in the size of the temporal specification. Even if the states of the automaton are constructed when needed (i.e., on-the-fly) rather than in constructing the entire automaton in advance, then each state requires memory that can grow exponentially with the size of the property (essentially, a set of sets of subformulas).

The formula φ below has length quadratic in n. It monitors a sequence of the symbols **0**, **1**, \$ and #. Adjacent *blocks* of **0**s and **1**s are of some length n and are separated by #, except for the last block, which is separated from the previous one by \$. This last block needs to be identical with some block that appeared before. We denote by $\bigcirc^i$ a sequence of i occurrences of $\bigcirc$ in an LTL formula.

$$((\# \wedge \Box((\# \vee \$) \rightarrow (\wedge_{1 \leq i \leq n}(\bigcirc^i(0 \vee 1))) \wedge \bigcirc^{n+1}(\# \vee \$)) \wedge (\Diamond \$ \wedge \bigcirc \Box \neg \$)) \wedge$$
$$\Diamond(\# \wedge \wedge_{1 \leq i \leq n}((\bigcirc^i 0 \wedge \Box(\$ \rightarrow \bigcirc^i 0)) \vee (\bigcirc^i 1 \wedge \Box(\$ \rightarrow \bigcirc^i 1)))))$$

With blocks of size n, one can encode 2^n different sequences. After seeing the first \$ symbol, we have seen a subset of these many possibly sequences. Thus, we must remember the subset of sequences we have seen before inspecting the last block that appears after the \$. Encoding a single set of such sequences requires space of $O(2^n)$ (each possible sequence may appear or not appear). With less information than 2^n, there will be two prefixes with different sets of occurring numbers, which will have the same memory representation; this means that runtime verification will not be able to check the execution correctly.

The number of possibilities of sequences is $O(2^{2^n})$, and an automaton that represents the required property is hence doubly exponential in the size of n. We do not need to construct such an automaton in advance, and can calculate the subsets while monitoring the sequence, hence we need memory and time of $O(2^n)$.

4.2 Algorithm 2: Checking availability of future verdicts

We alter the above runtime verification algorithm to check whether positive or negative verdicts can still be obtained after the current monitored prefix at runtime.

We first present an algorithm that will identify when a *good* state cannot be reached anymore during monitoring.

1. Construct the automaton $\mathcal{B}_{\varphi}$ as in the previous algorithm.

2. Apply Depth First Search (DFS) from the accepting states *backwards* (contrary to the direction of the transitions), to check for states from which accepting states can be reached. Let S be the states from which one *cannot* reach an accepting state.
3. Replace the states in S with a single state $\bot$ with a self loop, obtaining the automaton $\mathcal{C}_\varphi$.
4. Follow the states of $\mathcal{C}_\varphi$ while monitoring an execution: Start with its initial state and progress from a state to a state according to the input events from the monitored execution. $\bot$ is reached exactly when there cannot be a *good* prefix anymore, i.e., a positive ("accept") verdict cannot be issued anymore for φ.

A symmetric algorithm checks when a *bad* state cannot be reached anymore. We perform depth first search on $\mathcal{B}_{\neg\varphi}$ to find all the states in which the accepting state is not reachable, then replace them by a single state $\top$ with a self loop, obtaining $\mathcal{C}_{\neg\varphi}$. Reaching $\top$ after monitoring a prefix means that we will not be able again to have a *bad* prefix, hence a negative ("failed") verdict cannot be issued anymore for φ.

We can perform runtime verification while updating synchronously the state of both automata, $\mathcal{C}_\varphi$ and $\mathcal{C}_{\neg\varphi}$ on-the-fly, upon each input event to check whether any (positive or negative) verdict can still be reached.

The automata constructed in this algorithm, $\mathcal{C}_\varphi$, and $\mathcal{C}_{\neg\varphi}$, can have a number of states that is doubly exponential in the size of φ. Alternatively, one can avoid the a priori construction of these automata [31] using a binary search on their state space. However, this makes then the incremental calculation between successive monitored events become doubly exponential in time in the size of φ. For RV to be able to work online, the incremental complexity is critical and this is hardly reasonable. Hence, a pre-calculation of these two automata, before the monitoring starts, which leaves the incremental time complexity exponential in φ, as in Algorithm 1, is preferable.

4.3 Algorithm 3: Checking monitorability

A small variant on the construction of $\mathcal{C}_\varphi$ and $\mathcal{C}_{\neg\varphi}$ allows checking if a property is monitorable. The algorithm is simple: construct the product $\mathcal{C}_\varphi \times \mathcal{C}_{\neg\varphi}$ and check whether the state $(\bot, \top)$ is reachable. If so, the property is non-monitorable, since there is a prefix that will transfer the product automaton to this state and thus it is *ugly*. It is not sufficient to check separately that $\mathcal{C}_\varphi$ can reach $\top$ and that $\mathcal{C}_{\neg\varphi}$ can reach $\bot$.

In the property $(\Box\neg(p \wedge r) \wedge ((\neg p\,\mathcal{U}(r \wedge \Diamond q)) \vee (\neg r\,\mathcal{U}(p \wedge \Box q))))$: both $\bot$ and $\top$ can be reached, separately, depending on which of the predicates r or p happens first. But in either case, there is still a possibility for a *good* or a *bad* extension, hence it is a monitorable property. Specifically, if r holds in the monitored execution before p, then only a *good* prefix can happen, and if p happens before r, only a bad prefix can happen (if p and r holds simultaneously, a *bad* prefix is reported).

If the automaton $\mathcal{C}_\varphi \times \mathcal{C}_{\neg\varphi}$ consists of only a single state $(\bot, \top)$, then there is no information whatsoever that we can obtain from monitoring the property.

The above algorithm is simple enough to construct, however its complexity is doubly exponential in the size of the given LTL property. This may not be a problem, as the algorithm is performed off-line and the LTL specifications are often quite short.

Theorem 1 *Deciding monitorability is in EXPSPACE-complete.*

Proof. The upper bound is achieved by a binary search version of this algorithm[4]. For the lower bound we show a reduction from checking if a property is (not) a *liveness* property, a problem known to be in EXPSPACE-complete [23, 34].

First, let us establish that if ψ is satisfiable, then $\Diamond\psi$ is monitorable (i.e., every finite sequence can be extended into a *good* or *bad* sequence) iff ψ has a *good* prefix. To see this, observe that if ψ has a *good* prefix ρ, then any finite sequence σ can be extended to a *good* prefix $\sigma\rho$ of $\Diamond\psi$, hence $\Diamond\psi$ is monitorable. Lets consider now the other direction. Since ψ is assumed to be satisfiable, $\Diamond\psi$ cannot have a bad prefix, since we can extend any finite sequence by a sequence satisfying ψ in order to satisfy $\Diamond\psi$. Thus, if $\Diamond\psi$ is monitorable, then this is due to the existence of *good* prefixes. Clearly a *good* prefix of $\Diamond\psi$ has a suffix that is a good sequence of ψ.

By definition, ψ has a good prefix iff ψ is not in the *morbidity* class of properties. Then from the previous paragraph, if ψ is satisfiable, then $\Diamond\psi$ is monitorable iff ψ is not *morbidity*. We also know that ψ is *morbidity* iff $\neg\psi$ is not *liveness*. So, if ψ is satisfiable, then $\Diamond\psi$ is monitorable iff $\neg\psi$ is not *liveness*. Let $\varphi = \neg\psi$. Then, we have established that if $\neg\varphi$ is satisfiable (φ is not a tautology), then $\Diamond\neg\varphi$ is monitorable iff φ is not *liveness*.

We hence can check if φ is *liveness* as follows: first check if it is a tautology. This can be done in PSPACE (see [35]). If so, it is *liveness*. Otherwise, check if $\Diamond\neg\varphi$ is not monitorable. This establishes a reduction from a subset of the monitorability problem to *liveness*. Thus, monitorability cannot be easier than EXPSPACE. □

4.4 Algorithm 4: Identifying the class of a property

We can identify the classes of properties AFS (*guarantee*), SFS, NFS (*morbidity*), AFR (*safety*), SFR and NFR (*liveness*) for any given temporal property. Thus, we can also identify if a property is in an intersection of two of these classes.

For the classes AFS, SFS and NFS, we reverse acceptance in $\mathcal{C}_\varphi$, i.e., all states are accepting except for the empty state, obtaining $\widehat{\mathcal{C}}_\varphi$. We take now the product $\widehat{\mathcal{C}}_\varphi \times \mathcal{A}_\varphi$ and check its emptiness. We can apply a procedure that performs model checking with the property φ and the state space of $\widehat{\mathcal{C}}_\varphi$, see [9]. The language (accepted sequences) of $\widehat{\mathcal{C}}_\varphi \times \mathcal{A}_\varphi$ consists exactly of the executions that satisfy the property φ and do not have a *good* prefix. For such executions it is never sufficient to observe a finite prefix in order to decide that the property is satisfied. We apply a similar construction for AFR, SFR, NFR, removing the accepting state from $\mathcal{C}_{\neg\varphi}$ to obtain $\mathcal{D}_{\neg\varphi}$, and taking the product $\widehat{\mathcal{C}}_{\neg\varphi} \times \mathcal{A}_{\neg\varphi}$.

We then have the following conditions for identifying the different classes:

- AFR (*safety*): $\widehat{\mathcal{C}}_{\neg\varphi} \times \mathcal{A}_{\neg\varphi} = \emptyset$.
  Because in this case, executions satisfying $\neg\varphi$, i.e., not satisfying φ, cannot avoid having a *bad* prefix.

[4] To show that a property is not monitorable, one needs to guess a state of $\mathcal{B}_\varphi \times \mathcal{B}_{\neg\varphi}$ and check that (1) it is reachable, and (2) one cannot reach from it an empty component, both for $\mathcal{B}_\varphi$ and for $\mathcal{B}_{\neg\varphi}$. (There is no need to construct $\mathcal{C}_\varphi$ or $\mathcal{C}_{\neg\varphi}$.)

- NFR (*liveness*): The automaton $\mathcal{C}_{\neg\varphi}$ consists of a single state $\top$.
  Because the automaton $\mathcal{C}_{\neg\varphi}$ consists of a single state $\top$ exactly when we will never observe a *bad* prefix.
- SFR: $\widehat{\mathcal{C}}_{\neg\varphi} \times \mathcal{A}_{\neg\varphi} \neq \emptyset$ and $\mathcal{C}_{\neg\varphi}$ does not consist of a single state $\top$.
  Because in this case, there is an execution that avoids having any *bad* prefix, but there are still prefixes that are *bad*.
- AFS (*guarantee*): $\widehat{\mathcal{C}}_{\varphi} \times \mathcal{A}_{\varphi} = \emptyset$.
  Because in this case, executions satisfying φ cannot avoid having a *good* prefix.
- NFS (*morbidity*): The automaton $\mathcal{C}_{\varphi}$ consists of a single state $\bot$.
  Because the automaton $\mathcal{C}_{\varphi}$ consists of a single state $\bot$ exactly when we can never observe a *good* prefix.
- SFS: $\widehat{\mathcal{C}}_{\varphi} \times \mathcal{A}_{\varphi} \neq \emptyset$ and $\mathcal{C}_{\varphi}$ does not consist of a single state $\bot$.
  Because in this case, there is an execution that avoids having any *good* prefixes, but there are still prefixes that are *good*.

For a more efficient algorithm for checking if an LTL formula is a *safety* (AFR) see [34]. There, an algorithm, based on a binary search on the construction of $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\varphi}$ is presented. That algorithm is polynomial space in the size of the property φ. Hence the problem of checking *safety* is in PSPACE. A lower bound, showing that the problem is in PSPACE-complete is also given in [34]: one can check whether φ is valid (a problem known to be in PSPACE-complete) exactly when $\varphi \vee \Diamond p$ is a *safety* property, where p is a proposition that does not appear in φ. Thus, the same result applies to checking if an LTL formula is a *guarantee* property.

Checking *liveness* (NFR) was shown to be in EXPSPACE-complete in [23]. Thus, checking that a property is in SFR is also in EXPSPACE-complete, since SFR complements AFR $\cup$ NFR, hence is equivalent to checking that the property is neither *safety*, nor *liveness*. For the same reasons, these complexity results also apply to the dual classes: by checking the negation of the given property, we have that *guarantee* (AFS) is in PSPACE-complete, and that *morbidity* (NFS) and SFS are in EXPSPACE-complete. This agrees with the complexity of the binary search based algorithms given above.

4.5 Monitoring *Safety* and *Guarantee* Properties

Manna and Pnueli [27] identified the LTL *safety* properties with $\Box$LTL and the *guarantee* properties with $\Diamond$LTL. Runtime verification of temporal specifications in many cases concentrates on the past portion of the logic, and specifically on $\Box$LTL. Past time specifications have the important characteristics that one can distinguish when they are violated after observing a finite prefix of an execution. For an extended discussion of this issue of *monitorability*, see e.g., [4, 14].

The RV algorithm for $\Box$LTL, presented in [19], is based on the observation that the semantics of the past time formulas $\ominus\varphi$ and $(\varphi \mathcal{S} \psi)$ in the current state i is defined in terms of the semantics of its subformula(s) in the previous state $i-1$. To demonstrate this, we rewrite the semantic definition of the $\mathcal{S}$ operator to a form that is more applicable for runtime verification.

- $(\sigma, i) \models (\varphi \mathcal{S} \psi)$ if $(\sigma, i) \models \psi$ or: $i > 1$ and $(\sigma, i) \models \varphi$ and $(\sigma, i-1) \models (\varphi \mathcal{S} \psi)$.

The semantic definition is recursive in both the length of the prefix and the structure of the property. Thus, subformulas are evaluated based on smaller subformulas, and the evaluation of subformulas in the previous state. The algorithm shown below uses two vectors of values indexed by subformulas: pre, which summarizes the truth values of the subformulas for the execution prefix that ends just *before* the current state, and now, for the execution prefix that ends with the current state. The order of calculating now for subformulas is bottom up, according to the syntax tree.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) := \mathit{false}$.
2. Observe a new event (as a set of propositions) s as input.
3. Let $\mathsf{pre} := \mathsf{now}$.
4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \mathit{true}$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathsf{now}(\varphi)$ *and* $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\neg\varphi) :=$ *not* $\mathsf{now}(\varphi)$.
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) := \mathsf{now}(\psi)$ *or* $(\mathsf{now}(\varphi)$ *and* $\mathsf{pre}((\varphi \mathcal{S} \psi)))$.
   - $\mathsf{now}(\ominus \varphi) := \mathsf{pre}(\varphi)$.
5. If $\mathsf{now}(\eta) = \mathit{false}$ then report a violation, otherwise goto step 2.

Besides its simplicity, compared with the LTL monitoring algorithm in Section 4.1, the $\Box$LTL algorithm also has a linear complexity in the size of the specification. However, the fact that the algorithm is linear needs to be taken with a grain of salt. For consider the property φ expressed in LTL from Section 4.1; it was used to show that the memory and the time complexity that is required for performing even a single step of runtime verification for this problem is exponential in n while the specification is only quadratic in n. Now, since (1) this must be the complexity irrespective of the way the property φ is written, and (2) the monitoring algorithm is linear in the size of the $\Box$LTL specification, we can deduce that the $\Box$LTL specification itself, unlike φ, must be exponential in the size of n.

Now, monitoring a $\Box$LTL property $\Box\varphi$ can be done using the above algorithm. Monitoring $\Diamond$LTL can be performed similarly, only that a positive verdict is annouced once φ holds for the first time.

5 Conclusion

Temporal specification is often focused on infinite execution sequences. This abstracts the idea that the correctness requirements for a system should not depend on its bounded execution. Although model checking is capable of checking such properties for finite state systems, one can never exhaustively test an infinite execution. Runtime verification offers an alternative approach to model checking. It can be applied directly to the system itself, and it can help with testing the system when its state space size is prohibitively high, or monitor the system in deployment. On the other hand, runtime verification is limited to observing at any point only a finite portion of the execution.

The notion of monitorability identifies the kinds of verdicts that one can obtain from observing finite prefixes of an execution. Monitorability deals with the ability to obtain a verdict, positive or negative, given a finite prefix of an execution. In particular, non-monitorability characterizes situations where it may not be worthy anymore to wait for a verdict. However, we argued that the definition of monitorability needs to be refined, allowing to monitor properties where a priori there are some useful verdicts that may be observed, even if after observing some prefix of the execution these verdicts are not available anymore.

We studied here the connection between monitorability and Lamport's classification of properties as *safety* and *liveness*. To do that we needed to extend this classification using the dual classes, *guarantee* and *morbidity*, and complete the picture with another class that we termed *quaestio*.

We also provided algorithms for checking whether a property is monitorable or not, whether it belongs to a certain monitorability class, and what kind of verdict (positive or negative) we can expect after monitoring a certain prefix against a given property. This is useful to decide whether one should apply runtime verification for a given temporal property given expected verdicts, and what kind of verdicts one can still obtain after a given monitored prefix. It also allows to recognize when, during runtime verification, there is no further interesting information that we can expect, consequently abandoning the monitoring.

Acknowledgments. The authors would like to thank Moran Omer for useful comments on the manuscript. The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by Doron Peled was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

References

[1] B. Alpern, F. B. Schneider, Recognizing safety and liveness. Distributed Computing 2(3): 117-126, 1987.

[2] E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, G. Reger, An introduction to runtime verification, lectures on runtime verification - introductory and advanced topics, LNCS Volume 10457, Springer, 1-23, 2018.

[3] D. A. Basin, C. C.i Jiménez, . Klaedtke, E. Zalinescu, Deciding safety and liveness in TPTL. Information Processing Letters 114(12), 680-688 (2014).

[4] A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly?, RV'07, LNCS Volume 4839, Springer, 126-138, 2007.

[5] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL. ACM Trans. Software Engineering Methodologies, 20(4): 14:1-14:64, 2011.

[6] R. Bloem, B. Könighofer, R. Könighofer, C. Wang: Shield synthesis: - runtime enforcement for reactive systems. TACAS 2015: 533-548.

[7] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: application to model-checking. CONCUR 1997: 135-150

[8] E. M. Clarke, E. A. Emerson: Design and synthesis of synchronization skeletons using branching-time temporal logic. Logic of Programs 1981: 52-71.

[9] E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT Press, 2000.

[10] V. Diekert, M. Leucker, Topology, monitorable properties and runtime verification. Theoretical Computer Science 537: 29-41 (2014).

[11] O. Drissi-Kaitouni, C. Jard, Compiling temporal logic specifications into observers, INRIA Research Report RR-0881, 1988.

[12] E. A. Emerson, E. M. Clarke, Characterizing correctness properties of parallel programs using fixpoints. ICALP 1980: 169-181.

[13] Y. Falcone, J.-C. Fernandez, L. Mounier, Runtime verification of safety/progress properties, RV'09, LNCS Volume 5779, Springer, 40-59, 2009.

[14] Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime? STTT 14(3), 349-382, 2012.

[15] J.-C. Fernandez, C. Jard, T. Jéron, C. Viho, An experiment in automatic generation of test suites for protocols with verification technology. Sci. Comput. Program. 29(1-2), 123-146, 1997.

[16] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification, Engineering Dependable Software Systems 2013, IOS Press, 141-175

[17] R. Gerth, D. A. Peled, M. Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic. PSTV 1995: 3-18.

[18] K. Havelund, G. Reger, D. Thoma, and E. Zălinescu, Monitoring events that carry data, lectures on runtime verification - introductory and advanced topics, LNCS Volume 10457, Springer, 61-102, 2018.

[19] K. Havelund, G. Rosu, Synthesizing monitors for safety properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.

[20] M. Isberner, F. Howar, B. Steffen, The TTT algorithm: a redundancy-free approach to active automata learning, RV'14, LNCS Volume 8734, Springer, 307-322, 2014.

[21] M. Isberner, F. Howar, B. Steffen, Learning register automata: from languages to program structures, Mach. Learn. 96(1-2), Kluwer Academic Publishers, 65-98, 2014.

[22] M. Isberner, F. Howar, B. Steffen, The open-source LearnLib, CAV'15, LNCS Volume 9206, Springer, 487-495, 2015.

[23] O. Kupferman, G. Vardi, On relative and probabilistic finite counterability. Formal Methods in System Design 52(2): 117-146, 2018.

[24] O. Kupferman, M. Y. Vardi, Model checking of safety properties. Formal Methods in System Design 19(3): 291-314, 2001.

[25] L. Lamport, Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. 3(2): 125-143, 1977.

[26] K. G. Larsen, A. Legay, Statistical model checking: past, present, and future, ISoLA'16, LNCS Volume 9953, Springer, 3-15, 2016.

[27] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification. Springer, 1992.

[28] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, STTT, Springer, 249-289, 2011.

[29] D. Peled, K. Havelund, Refining the safety-liveness classification of temporal properties according to monitorability. Models, Mindsets, Meta 2018: 218-234.

[30] D. A. Peled, M. Y. Vardi, M. Yannakakis, Black box checking, FORTE/P-STV'99, IFIP Conference Proceedings Volume 156, Kluwer, 225-240, 1999.

[31] A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers. FM'06, LNCS Volume 4085, Springer, 573-586, 2006.

[32] J.-P. Queille, J. Sifakis, Iterative methods for the analysis of Petri nets. Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets 1981: 161-167.

[33] S. Safra, On the complexity of omega-automata, FOCS 1988, 319-327.

[34] A. P. Sistla, Safety, liveness and fairness in temporal logic, Formal Aspects of Computing 6(5): 495-512, 1994.

[35] A. P. Sistla, E. M. Clarke, The complexity of propositional linear temporal logics, STOC 1982: 159-168.

[36] W. Thomas, Automata on infinite objects, handbook of theoretical computer science, Volume B: Formal Models and Semantics, 133-192, 1990.

[37] M. Y. Vardi, P. Wolper, Automata-theoretic techniques for modal logics of programs, Journal of Computer System Science 32(2): 183-221, 1986.