Refining the Safety–Liveness Classification of Temporal Properties According to Monitorability

Doron Peled[1(✉)] and Klaus Havelund[2(✉)]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
doron.peled@gmail.com

[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
klaus.havelund@jpl.nasa.gov

Abstract. Runtime verification is the topic of analyzing execution traces using formal techniques. It includes monitoring the execution of a system against temporal properties, commonly to detect violations. Not every temporal property is fully monitorable however: in some cases, the correctness of the execution does not depend on any finite prefix. We study the connection between monitorability and Lamport's classification of properties to safety and liveness and their dual classes. We refine the definition of monitorability and provide algorithms to check which verdicts can be expected, a priori and during runtime verification.

1 Introduction

Runtime verification facilitates the direct monitoring of the execution of a system, checking it against a formal specification. This can be useful for many applications, including testing a system before it is deployed, as well as monitoring the system after deployment. This approach 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.

Monitoring properties are often given in linear temporal logic (LTL) [22]. These properties are traditionally interpreted over infinite execution sequences (the monitored system keeps emitting events). But for runtime verification to

D. Peled—The research performed by this author was partially funded by Israeli Science Foundation grant 2239/15: "Runtime Measuring and Checking of Cyber Physical Systems".

K. Havelund—The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

be useful, it is necessary to be able to provide information after observing only finite execution sequences, also referred to as *prefixes*. For example, the property $\Box p$ (for some atomic proposition p), which asserts that p always happens, can be refuted by a runtime monitor 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, resulting in a negative verdict. However, no finite prefix of an execution can establish that $\Box p$ holds. In a similar way, the property $\Diamond p$ cannot be refuted, since p may appear at any time in the future; but once p happens, we know 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, in the case 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; at this point we can abandon the monitoring, since the property cannot be further violated. On the other hand, for the property $\Box\Diamond p$ we can never provide a verdict in finite time: for whatever happens, p can still appear an infinite number of times, and we cannot guarantee or refute that this property holds when observing any finite prefix of an execution. The problem of monitorability of a temporal property was studied in [5,10,25], basically requiring that at any point of monitoring we still have a possibility to obtain a finite positive or negative verdict.

We refine here the study of LTL monitorability, distinguishing cases where *some* verdicts are always possible during runtime, *no* verdicts are expected, or some verdicts are possible *a priori*, but may not be available later, depending on the monitored prefix. We extend Lamport's safety and liveness classification of temporal properties with guarantee, which is the dual of safety, 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, or morbidity property.

Related Work. Alpern and Schneider [1] formalized Lamport's definition of safety and liveness, Sistla [27] showed a PSPACE algorithm for checking safety, and an EXPSPACE algorithm for checking liveness. Checking liveness was shown to be in EXPSPACE-complete in [18]. Drissi-Kaitouni and Jard [8], as well as Kupferman and Vardi [19] studied the problem of monitoring LTL properties for an execution sequence. Pnueli and Zaks [25] 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 [11]. Bauer, Leucker and Schallhart 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 [7] studied monitorability and its connection to safety and liveness using topological characterizations. Falcone, Fernandez and Mounier [9] 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 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.

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

2 Preliminaries

2.1 Runtime Verification

Runtime verification (RV) [2,13] very generally refers to the use of rigorous (formal) techniques for *processing* execution traces emitted by a system being observed. The purpose is, again generally viewed, 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. Nonetheless, RV can be useful due to the rigorous methods involved. Note that 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 course 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 the monitored system.

Processing can take numerous forms. We focus here on *specification-based* runtime verification, where an execution trace is checked against a property expressed in a formal (usually temporal) logic. 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. Each event e_i captures a snapshot of S's execution state. Assume the type $\mathbb{E}$ of events; then the RV problem can be formulated as constructing a program $M : \mathbb{E}^* \rightarrow D$, 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 with the specification.

However, the field 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 [15–17,24] and statistical model checking [21], giving M the type $M : 2^{\mathbb{E}^*} \rightarrow D$.

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 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 operators [22]:

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

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 $(\varphi \vee \psi)$ instead of $\neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi)$ instead of $(\neg\varphi \vee \psi)$, $\Diamond\varphi$ (*eventually* φ) instead of $(\mathit{true}\,\mathcal{U}\,\varphi)$ and $\Box\varphi$ (*always* φ) instead of $\neg\Diamond\neg\varphi$.

LTL formulas are interpreted over an infinite sequence of events[1] $\xi = e_0.e_1.e_2\ldots$, where $e_i \subseteq P$ for each $i \geq 0$. These are the propositions that *hold* in that event. We denote by ξ_i the suffix $e_i.e_{i+1}.e_{i+2}\ldots$ of ξ. LTL semantics is defined as follows:

– $\xi_i \models \mathit{true}$.
– $\xi_i \models p$ iff $p \in e_i$.
– $\xi_i \models \neg\varphi$ iff not $\xi_i \models \varphi$.
– $\xi_i \models (\varphi \wedge \psi)$ iff $\xi_i \models \varphi$ and $\xi_i \models \psi$.
– $\xi_i \models \bigcirc\varphi$ iff $\xi_{i+1} \models \varphi$.
– $\xi_i \models (\varphi \,\mathcal{U}\, \psi)$ iff for some $j \geq i$, $\xi_j \models \psi$, and for all $i \leq k < j$, $\xi_k \models \varphi$.

Then $\xi \models \varphi$ when $\xi_0 \models \varphi$.

An LTL property can be translated into a nondeterministic Büchi automaton [12,30]. The translation can incur an exponential blowup. This nondeterministic automaton can be used directly for model checking, but requires determinization [26], e.g., for the purpose of synthesizing a reactive system from the temporal property. Unfortunately, determinization results here in additional exponential explosion. This sums up to a double exponential blowup of the translation from the LTL property to the deterministic (Rabin, Street) automaton that accepts the same language. It turns out that we also need (a different kind of) determinization for runtime verification [19].

Past time LTL (PLTL) is interpreted over finite sequences, looking backwards from the current event. PLTL has the back mirror operators of LTL's modal operators.

Bauer, Leucker and Schallhart [5] define three categories of prefixes of execution sequences over 2^P for a temporal property φ.

– A *good* prefix is one where all its extensions (infinite sequences of elements from 2^P) satisfy 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.

[1] The classical interpretation of LTL is over states [22], but in the context of RV, we monitor a sequence of events that are reported by the instrumentation.

3 Characterizing Temporal Properties

Safety and liveness temporal properties were defined informally on infinite execution sequences by Lamport [20] 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 [22]. 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 prefix can be extended to satisfy φ.

morbidity (co-liveness) A property φ is a *morbidity* property if every finite prefix can be extended to violate φ.

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 [4,23]. This motivates providing three kinds of verdicts:

failed when the current prefix cannot be extended in any way into an execution that satisfies the specification,

satisfied 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, we can provide an indication as soon as it fails. Correspondingly, we can report on the satisfaction of a *guarantee* property as soon as a finite prefix satisfies it. The only property that is both a safety and a liveness (and a guarantee) property is *true*.

Each temporal property is a conjunction of a liveness and a safety property [1]. Due to the connection between safety and guarantee and between liveness and morbidity, we immediately obtain that every temporal property is a disjunction of a guarantee and a morbidity property. Manna and Pnueli characterized syntactically the temporal safety properties as $\Box\varphi$, and the guarantee properties as $\Diamond\varphi$, where φ is a PLTL 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 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.

AFR (safety) Always Finitely Refutable: when the property does not hold on an infinite execution, refutation can be identified after a finite (bad) prefix.
AFS (guarantee) Always Finitely Satisfiable: when the property is satisfied on an infinite execution, satisfaction can be identified after a finite (good) prefix.
NFR (liveness) Never Finitely Refutable: Refutation (i.e., a bad prefix) can never be identified after a finite prefix.
NFS (morbidity) Never Finitely Satisfiable: Satisfaction (i.e., a good prefix) can never be identified after a finite prefix.

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 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 prefix; for other infinite executions satisfying the property, this is not the case.

Let *Prop* be the set of all properties expressible in some temporal formalism, e.g., LTL or Büchi automata. Then it is clear that *Prop* = AFR ∪ SFR ∪ NFR. The only property that is mutual to two of these classes is *true*, which holds both for AFR and for NFR. It also holds that *Prop* = AFS ∪ SFS ∪ NFS. The only temporal property that is mutual to two of these classes (AFS and NFS) is *false*. 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 sub-classes as shown in Fig. 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 in Fig. 1.

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

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

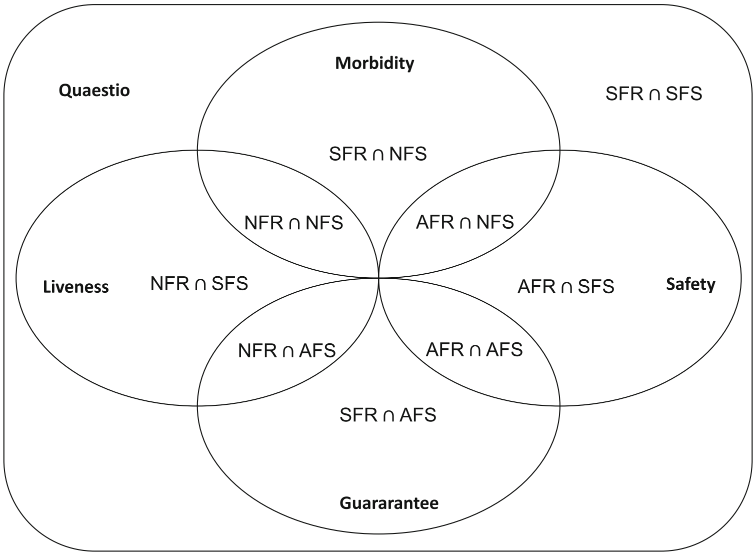

Fig. 1. Classification of properties: safety, guarantee, liveness, morbidity and quaestio.

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

Observe that for AFR $\cap$ AFS we gave an example of a property with only the nexttime operator $\bigcirc$. We show that for LTL, any property φ in AFR $\cap$ 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[2] 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. So assume that this path satisfies 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,

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

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 paths in the tree using $\bigcirc$ and the Boolean operators[3]. 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 a safety and liveness.

4 Monitorability

4.1 Defining Monitorability

The *good*, *bad* and *ugly* prefixes, defined in [5] and presented at the end of Sect. 2.2, are related to the ability to provide a verdict about a temporal property when monitoring an execution trace. When identifying a good or a bad finite prefix, we are done 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 [25].

Ugly prefixes cannot occur in an execution satisfying a safety property [5]. Suppose by contradiction that there is an ugly prefix σ for a safety property φ. Note that if a prefix is ugly, it cannot have a good or a bad prefix. Now extending an ugly prefix σ in any way into an execution that does not satisfy a safety property φ entails that there must be a bad (finite) prefix extending σ, a contradiction to σ being ugly. So, any infinite extension of σ must satisfy φ. But then σ itself must be a good prefix, a contradiction again to σ being ugly. 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.

4.2 Runtime Verification Algorithms for Monitorability

We present four algorithms. The first one is a classical algorithm for runtime verification of LTL (or Büchi automata) properties. The second algorithm can be used to check during run time what kind of verdicts can still be produced given the current prefix. The third algorithm can be used to check whether the property is monitorable, and also be used under the refinement of monitorability that we present in the next section. The fourth algorithm can be used to check the class of a given temporal property under the extension of Lamport's safety/liveness characterization given in this paper.

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

Algorithm 1. Monitoring Sequences Using Automata

Kupferman and Vardi [19] provide an algorithm for detecting good and bad prefixes. For good prefixes, start by constructing a Büchi automaton $\mathcal{A}_{\neg\varphi}$ for $\neg\varphi$, e.g., using the translation in [12]. Note that this automaton is not necessarily deterministic [29]. States of $\mathcal{A}_{\neg\varphi}$, from which one cannot reach a cycle that contains an accepting state, are deleted. Checking for a positive verdict for φ, one keeps for each monitored prefix the set of states that $\mathcal{A}_{\neg\varphi}$ would be after observing that input. One starts with the set of initial states of the automaton $\mathcal{A}_{\neg\varphi}$. Given the current set of successors S and an event $e \in 2^P$, 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, $S' = \{s' | s \in S \land (s, e, s') \in \Delta\}$. 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* for a deterministic automaton $\mathcal{B}_\varphi$, whose initial state is the set of initial states of $\mathcal{A}_{\neg\varphi}$, accepting state is the empty set, and transition relation as described above. The size of this automaton is $\mathcal{O}(2^{2^{|P|}})$, resulting in double exponential explosion from the size of the checked LTL property. But in fact, we do not need to construct the entire automaton $\mathcal{B}_\varphi$ in advance, and can avoid the double exponential explosion by calculating its current state on-the-fly, while performing runtime verification. Thus, the incremental processing per each event is exponential in the size of the checked LTL property. Unfortunately, a single exponential explosion is unavoidable [19].

Checking for a *failed* verdict for φ is done with a symmetric construction, 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 as 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 *failed* verdict if $\mathcal{B}_{\neg\varphi}$ reaches an accepting state, a *satisfied* 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.

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. Applying DFS on $\mathcal{B}_\varphi$, we search for states from which one cannot reach the accepting state. Then we replace these states with a single state $\bot$ with a self loop, obtaining the automaton $\mathcal{C}_\varphi$. Reaching $\bot$, after monitoring a finite prefix σ with $\mathcal{C}_\varphi$ happens exactly when we will not have a good prefix anymore. This means that after σ, a *satisfied* verdict cannot be issued anymore for φ.

Similarly, we perform BFS 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 *failed* verdict cannot be issued anymore for φ.

We can perform runtime verification while updating the state of both automata, $\mathcal{C}_\varphi$ and $\mathcal{C}_{\neg\varphi}$ on-the-fly, upon each input event. However, we need to be able to predict if, from the current state, an accepting state is not reachable. While this can be done in space exponential in φ, it makes an incremental calculation whose time complexity is doubly exponential in the size of φ, as is the algorithm for that by Pnueli and Zaks [25]. This is hardly a reasonable complexity for the incremental calculation performed between successive monitored events for an on-line algorithm. Hence, a pre-calculation of these two automata before the monitoring starts is preferable, leaving the incremental time complexity exponential in φ, as in Algorithm 1.

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 $(\neg(p \wedge r) \wedge ((\neg p \mathcal{U}(r \wedge \Diamond q)) \vee (\neg r \mathcal{U}(p \wedge \Box q))))$: 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.

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.

We show that checking monitorability is in EXPSPACE-complete. 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 [18,27].

- We first neutralize bad prefixes. Now, when ψ is satisfiable, then $\Diamond\psi$ is monitorable (specifically, any prefix can be completed into a *good* prefix) iff ψ has a good prefix.
- Checking satisfiability of a property ψ is in PSPACE-complete [28][5].
- ψ has a good prefix iff ψ is not a morbidity property, i.e., if $\varphi = \neg\psi$ is not a liveness property.
- Now, φ is *not* a liveness property iff either φ is unsatisfiable or $\Diamond\neg\varphi$ is monitorable.

[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 $\mathcal{C}_{\neg\varphi}$.).

[5] Proving that liveness was PSPACE-hard was shown in [3].

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 [6]. 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 state.

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 state, 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 state.

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 state, but there are still prefixes that are good.

For a more efficient algorithm for checking if an LTL formula is a safety (AFR) see [27]. 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 [27]: 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 [18]. 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.

4.3 Refining Monitorability

We first look at the relationship between the above classification of properties and monitorability. Any property that is in AFR (safety) or in AFS (guarantee) is monitorable as identified in [5,10]. A property that is NFR $\cap$ NFS is non-monitorable. In fact no verdict is ever expected on any sequence that is monitored against such a property. This leaves the three classes SFR $\cap$ SFS, SFR $\cap$ NFS and NFR $\cap$ SFS, for which some properties are monitorable and others are not. This is demonstrated in the following table.

Class	Monitorable example	Non-monitorable example
SFR $\cap$ SFS	$((\Diamond r \vee \Box\Diamond p) \wedge \bigcirc q)$	$((p \vee \Box\Diamond p) \wedge \bigcirc q)$
SFR $\cap$ NFS	$(\Diamond p \wedge \Box q)$	$(\Box\Diamond p \wedge \bigcirc q)$
NFR $\cap$ SFS	$(\Box p \vee \Diamond q)$	$(\Box\Diamond p \vee \bigcirc q)$

We propose that RV can still be applied for non-monitorable properties if initially some verdicts can be made. We refine the definition of monitorability into the following categories:

- A property is *monitorable* if it cannot have an ugly prefix. This corresponds to the definition of monitorability in [5,25]. Safety and guarantee properties are universally monitorable. But as demonstrated above, some of the properties in SFR $\cap$ SFS, SFR $\cap$ NFS and NFR $\cap$ SFS are also monitorable.

  Checking monitorability can be done using Algorithm 3. In Fig. 2, the light gray areas correspond to properties that are monitorable.
- 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. The black area in Fig. 2 correspond to properties with 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 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 check whether *some* verdict is still possible for the current prefix, abandoning the runtime verification when this is not the case. The dark gray areas in Fig. 2 represent the weakly monitorable properties.

Consider the property $(p \vee (\neg q\ \mathcal{U}\ (p \wedge \Box\Diamond r)))$. This property is in 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. Algorithm 3 can identify the fact that this property is both non-monitorable but is not a zero monitoring information property.

This calls for using Algorithm 2 rather than Algorithm 1 to perform the runtime verification. Suppose now that the first event is $\{q\}$. Since p does not hold in the first event, we still have to satisfy the right disjunct $(\neg q\, \mathcal{U} (p \wedge \Box\Diamond r))$. Algorithm 2 can inform that from now on, one can expect only a negative verdict. If the next event is $\{\}$, Algorithm 2 will inform that no further verdict can be given, hence monitoring can be aborted.

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 is prohibitively high. 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.

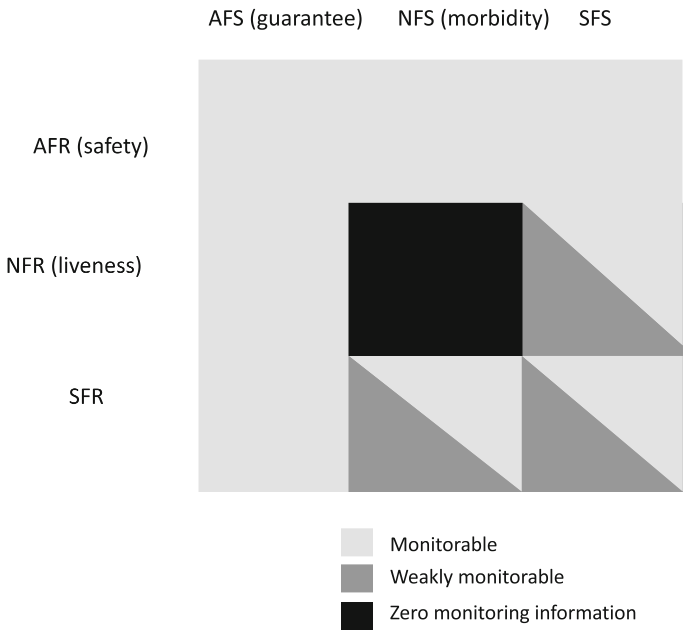

Fig. 2. Classification of properties according to monitorability.

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 property 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.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
2. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
3. Basin, D.A., Jiménez, C.C., Klaedtke, F., Zalinescu, E.: Deciding safety and liveness in TPTL. Inf. Process. Lett. **114**(12), 680–688 (2014)

4. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Method. **20**(4), 14:1–14:64 (2011)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
7. Diekert, V., Leucker, M.: Topology, monitorable properties and runtime verification. Theor. Comput. Sci. **537**, 29–41 (2014)
8. Drissi-Kaitouni, O., Jard, C.: Compiling temporal logic specifications into observers, INRIA Research Report RR-0881 (1988)
9. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_4
10. Falcone, Y., Fernandez, J.-C., Mounier, L.: What can you verify and enforce at runtime? STTT **14**(3), 349–382 (2012)
11. Fernandez, J.-C., Jard, C., Jéron, T., Viho, C.: An experiment in automatic generation of test suites for protocols with verification technology. Sci. Comput. Program. **29**(1–2), 123–146 (1997)
12. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembiński, P., Średniawa, M. (eds.) PSTV 1995. IFIPAICT, pp. 3–18. Springer, Boston (1996). https://doi.org/10.1007/978-0-387-34892-6_1
13. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
14. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
15. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
16. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Mach. Learn. **96**(1–2), 65–98 (2014)
17. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
18. Kupferman, O., Vardi, G.: On relative and probabilistic finite counterability. Formal Meth. Syst. Des. **52**(2), 117–146 (2018)
19. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Meth. Syst. Des. **19**(3), 291–314 (2001)
20. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
21. Larsen, K.G., Legay, A.: Statistical model checking: past, present, and future. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 3–15. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_1
22. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, New York (1992)

23. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. **14**, 249–289 (2011)
24. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems. IAICT, vol. 28, pp. 225–240. Springer, Boston, MA (1999). https://doi.org/10.1007/978-0-387-35578-8_13
25. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_38
26. Baier, C., Bertrand, N., Größer, M.: The effect of tossing coins in omega-automata. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 15–29. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04081-8_2
27. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects Comput. **6**(5), 495–512 (1994)
28. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. In: STOC 1982, pp. 159-168 (1982)
29. Thomas, W.: Automata on infinite objects, handbook of theoretical computer science. In: Formal Models and Semantics, vol. B, pp. 133–192 (1990)
30. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. Comput. Syst. Sci. **32**(2), 183–221 (1986)