Monitorability for Runtime Verification

Klaud Havelund[1]⋆ and Doron Peled[2]⋆⋆

[1]Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology
[2] Department of Computer Science, Bar Ilan University, Israel

Abstract. Runtime verification (RV) facilitates the formal analysis of execution traces. In particular, it permits monitoring the execution of a system and checking it against a temporal specification. Online RV observes, at any moment, a prefix of the complete monitored execution and is required to provide a verdict whether *all* the complete executions that share that prefix satisfy or falsify the specification. Not every property (and for every kind of verdict) lends itself to obtaining such an early verdict. Monitorability of a temporal property is defined as the ability to provide positive (success) or negative (failure) verdicts after observing a finite prefix of the execution. We classify temporal properties based on their monitorability and present related monitoring algorithms. A common practice in runtime verification is to concentrate on the class of *safety* properties, where a failure to satisfy the specification can always be detected in finite time. In the second part of the paper we concentrate on monitoring safety properties and their place among the other classes of properties in terms of algorithms and complexity.

1 Introduction

Runtime verification (RV) allows monitoring executions of a system, either online or offline, checking them against a formal specification. It can be applied to improve the reliability of critical systems, including safety as well as security aspects, and can more generally be applied for processing streaming information. RV is not a comprehensive verification method such as model checking [7, 12, 31], as it is applied separately to executions of the system one at a time. On the other hand, due to its more modest goal, RV lacks of some of the restrictions of more comprehensive formal methods related to complexity and applicability.

The specifications, against which the system is checked during RV, are often expressed in *linear temporal logic* (LTL) [26]. These properties are traditionally interpreted over infinite execution sequences. This corresponds to the case where the number of events that the monitored system can emit is unbounded[1]. Indeed, the input trace is

⋆ 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”.

[1] One can of course distinguish the case of terminating executions, or assume some indefinite padding by an *end-of-execution* event.

often a priori not limited to a specific length, and checking it against a given specification is supposed to follow it for as long as the monitored system is running. At any time, only a finite prefix of the system is observed. 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, consider the property $\Box p$ (for some atomic proposition p), which asserts that p always holds throughout the execution. A prefix of an execution can be *refuted* by a runtime monitor, i.e., demonstrating a *failure* to satisfy $\Box p$, if p does not hold in some observed event. At this point, no matter how the execution is extended, the property fails to hold. On the other hand, 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 $\Diamond p$, which asserts that p will eventually happen, cannot be refuted, since even if p has not happened yet, it may hold at any time in the future; on the other hand, once p holds, we have established that the property is satisfied, independent on any continuation, and we can issue a positive (*success*) verdict. For the property $\Box\Diamond p$ we can never provide a verdict in finite time, since for a finite prefix p can hold only finitely many times.

The *monitorability problem* of a temporal property was studied in [5, 14, 30]. Accordingly, a specification 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. Nevertheless, it is possible that a priori, or after some prefix, only one type of verdicts is possible.

We follow [29] in classifying temporal properties as an extension of Lamport's *safety* and *liveness* properties. The class *Guarantee* was defined to be the dual of *safety* in [26], i.e., the negation of a safety property is a guarantee property and vice versa. We then defined *morbidity* as the dual of *liveness*. To complete this classification to cover all possible temporal specification we added another class that we termed *quaestio*. 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.

The second part of the paper focuses on safety properties. In RV, one often expresses safety properties in the form $\Box\varphi$, where φ is a past LTL formula. Furthermore, it is often only the past property φ that is monitored, returning a *yes/no* verdict for the currently observed prefix. We describe the algorithms for monitoring such safety properties and compare them to the future LTL monitoring algorithm in terms of complexity. Monitoring propositional past formulas was extended to first order safety properties [18, 3]. In particular, this was focused on monitoring past-time first order LTL properties against traces that contain data. Although this resulted in quite efficient monitors, we show some theoretical limitations on monitoring first order LTL properties.

2 Preliminaries

2.1 Runtime Verification

Runtime verification [2, 19] 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 is devoid of some of the complexity and computational restrictions that some of the more comprehensive formal methods have. But on the other hand, RV does not provide a comprehensive coverage.

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 aspect-oriented programming frameworks. Processing of RV can take place online, as the system executes, or offline, by processing log files produced by the system. In the case of online processing, observations can be used to control (shield) the monitored system [6].

In specification-based runtime verification, an execution trace is checked against a property expressed in a formal (often temporal) logic or using automata notation. More formally, assume a finite prefix of an execution of an observed system up to a certain point is captured as an execution trace $\sigma = e_1.e_2.\ \ldots\ .e_n$, which is a sequence of observed events. Then the RV problem can be formulated as constructing a program, which when applied to the trace σ, returns some data value in a domain of interest D. In specification-based RV, the monitor is generated from a formal specification, given e.g. as a temporal logic formula, a state machine, or a regular expression. The domain of interest D is often the Boolean domain or some extension of it [4] (in particular adding a third value "?" for *yet unknown*) indicating whether the execution trace conforms to the specification.

The input trace for RV is typically observed one event at a time and the monitoring algorithm updates a *summary* that contains enough information to provide given verdicts without observing the previous events. This summary can be, e.g., a state in an automaton that implements the monitor, or a vector of Boolean values representing the subformulas of the specification that hold given the observed prefix. Updating the summary upon seeing a new event needs to be performed efficiently, in particular in online RV, to keep up with the speed of reported events. The complexity of updating the summary is called the *incremental complexity* and needs to be kept minimal. In particular, this complexity need not depend on the length of the prefixed observed so far, which can grow arbitrarily.

Monitored execution traces are 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. Monitorability focuses on the kind of verdicts that can be produced based finite prefixes given a specific property.

2.2 Linear Temporal Logic

The classical definition of (future) linear temporal logic is based on future modal operators [26] with the following syntax:

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

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 $false = \neg true$, $(\varphi \lor \psi) = \neg(\neg\varphi \land \neg\psi)$,

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

An event e consists of a subset of the propositions in P. These are the propositions that were observed to *hold* or to be *true* during that event. A *trace* $\sigma = e_1.e_2.e_3\ldots$ 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 define $\sigma \models \varphi$ when $\sigma, 1 \models \varphi$.

3 Monitorability

Online runtime verification observes at each point a prefix of the monitored execution sequence and provides a verdict against a specification. There are three kinds of verdicts:

– *failed* (or *refuted* or *negative*) when the current prefix cannot be extended in any way into an infinite execution that satisfies the specification. Then the current prefix is called a *bad* prefix [5].
– *satisfied* (or *established* or *positive*) when any infinite extension of the current prefix satisfies the specification. Then the current prefix is called a *good* prefix [5].
– *undecided* when the current prefix can be extended into an infinite execution that satisfies the specification but also extended into an infinite execution that satisfies its negation.

Undecided prefixes that cannot be extended to either a good or a bad prefix are called *ugly* [5], as no further monitoring information will be obtained by continuing the monitoring. As will be shown in Section 3.2, at the expense of a more complex algorithm, one can also decide and report when the current prefix is ugly.

Monitorability of a property φ is defined in [5] as the lack of *ugly* prefixes for the property φ. This requirement means that during monitoring, we never "lose hope" to obtain a verdict. This definition is consistent with an early definition in [30]. The definition of monitorability is a bit crude in the sense that it only distinguish between specifications for which during monitoring one can always still expect a verdict, and those for which this is not the case. But it lumps together specifications where only a positive verdict or only a negative verdict can be expected. We study here monitorability in a wider context, classifying the temporal properties into families according to the ability to produce particular verdicts.

3.1 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 defined by Manna and Pnueli [26]. We add to this classes the *morbidity* properties, which is the dual class of *liveness* properties. This leads us to the following classical way of describing these four classes of 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 violate 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.

Safety, *guarantee*, *liveness* and *morbidity* can be seen as characterizing different cases related to the 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, given in terms of the verdicts available for the different classes. 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. The four classes of properties mentioned above, however, do not cover the entire set of possible temporal properties, and we need to add two more classes to complete the classification.

– 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 in which 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.
– 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 one class of the form XFR, where X stands for A, S or N, and also to one class of the form XFS, again with X is A, S or N. The possible intersections between classes is shown in Figure 1. Below we give examples for the nine combinations of XFR and XFS, appearing in clockwise order according to Figure 1, ending with the intersection SFR∩SFS, termed *Quaetio* that appears in the middle.

– SFR ∩ NFS: $(\Diamond p \land \Box q)$
– AFR ∩ NFS: $\Box p$
– AFR ∩ SFS: $(p \lor \Box q)$
– AFR ∩ AFS: $\bigcirc p$
– SFR ∩ AFS: $(p \land \Diamond q)$
– NFR ∩ AFS: $\Diamond p$
– NFR ∩ SFS: $(\Box p \lor \Diamond q)$
– NFR ∩ NFS: $\Box\Diamond p$
– SFR ∩ SFS: $((p \lor \Box\Diamond p) \land \bigcirc q)$

Another way to cover all the temporal properties is as the union of *safety* (AFR), *guarantee* (AFS), *liveness* (NFR), *morbidity* (NFS) and *quaestio* (SFR∩SFS). 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.

The shadowed areas in Figure 1 in the intersections between the classes of properties NFS, SFS and the classes NFR, SFR correspond to the cases where monitorability is not guaranteed. While in NFR ∩ NFS there are no monitorable properties, in the other three intersections there are both monitorable and nonmonitorable properties. Examples for these cases appear in the following table.

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

3.2 Runtime Verification Algorithms for Monitorability

The following algorithm [24, 9] monitors executions and provides *success* (positive) or *fail* (negative) verdict of the checked property whenever a minimal *good* or a *bad* prefix is detected, respectively.

A procedure for detecting the minimal *good* prefix when monitoring an execution against the specification φ is as follows:

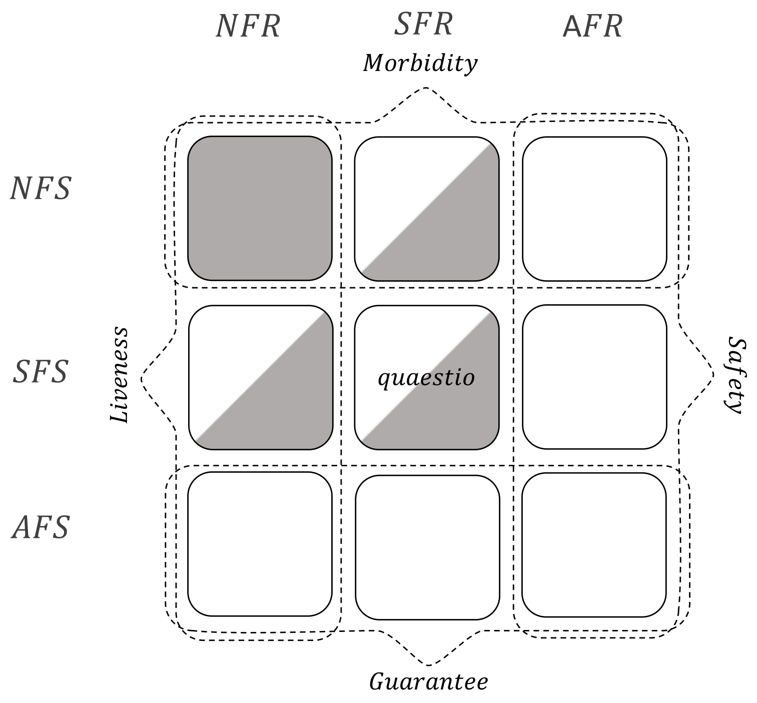

Fig. 1: Classification of properties according to monitorability: filled space correspond to nonmonitorable properties.

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 [35].
2. Using DFS, find the states of $\mathcal{A}_{\neg\varphi}$ from which one *cannot* reach a cycle that contains an accepting state and *remove* these states.
3. *On-the-fly subset construction:* While monitoring input events, maintain the current subset of states that the automaton $\mathcal{A}_{\neg\varphi}$ reaches after observing the current input as follows:
   - Start 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 set of successors S' contains 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 a positive verdict is issued. This is because the empty subset of states means that following the current inputs, the automaton $\mathcal{A}_{\neg\varphi}$ cannot complete the input into an accepting execution.

A symmetric procedure constructs $\mathcal{A}_{\varphi}$ for detecting minimal *bad* prefixes. One can monitor using both $\mathcal{A}_{\neg\varphi}$ and $\mathcal{A}_{\varphi}$ at the same time, providing both failure and success verdicts. Translating the formula φ into a Büchi automaton can result in an automaton $\mathcal{A}_\varphi$ of size $O(2^{|\varphi|})$. The subset construction described above has to keep in each state a set of states. Thus, the incremental complexity of the RV monitoring algorithm is also $O(2^{|\varphi|})$. The subsets of states that are constructed during monitoring form the *summary* of the trace observed so far, which is needed to continue the monitoring further.

Instead of the on-the-fly subset construction, one can precalculate, before monitoring, a deterministic automaton $\mathcal{B}$ based on the product $\mathcal{A}_\varphi \times \mathcal{A}_{\neg\varphi}$. Each state of the automaton is a pair of subsets of states, of the two automata, as constructed above. Then, each state of this automaton can be marked with $\bot$, $\top$ or ?, where $\bot$ corresponds to an empty subset of $\mathcal{A}_\varphi$ states (a *failed* verdict) and $\top$ corresponds to an empty set of $\mathcal{A}_{\neg\varphi}$ states (a *success* verdict). Instead of the on-the-fly updates in the above subset construction, monitoring can be performed while updating the state of the automaton based on the automaton $\mathcal{B}$. The size of this automaton is $O(2^{2^{|\varphi|}})$, but the size of each state remains $O(2^{|\varphi|})$, as in the on-the-fly version. Thus, the incremental complexity remains the same.

An advantage of the preliminary construction of the automaton $\mathcal{B}$ over the on-the-fly subset construction described above is that it can be further used to predict at runtime the kind of verdicts that can be expected after observing the current prefix. To allow this prediction, each state of $\mathcal{B}$ is annotated, during the preliminary construction, with the kind of verdicts, $\top$ (*success*), $\bot$ (*failed*) or both, that mark the nodes that are reachable from the current state. During monitoring, when neither verdicts is reachable anymore, the current prefix is identified as *ugly*. When the initial state of $\mathcal{B}$ is marked as *ugly*, the property is nonmonitorable.

3.3 A Lower Bound Example for LTL Monitoring

We present an example, following [24], to show that monitoring an LTL specification requires a summary of size exponential in the length of the property.

The specification is a safety property. It asserts about a nonempty and finite sequence of blocks of 0 and 1 bits of length n. Each block starts with the symbol #. Then, a final block, separated from the previous one by \$ follows. After the last block, the symbol & repeats indefinitely. The property asserts that if the trace has the above structure, the last block (the one after the \$) is identical to one of the blocks that appeared before. We denote by $\bigcirc^i$ a sequence of i occurrences of $\bigcirc$ in an LTL formula. The formula has length quadratic in n.

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

With n bits, one can encode 2^n different blocks. During the monitoring, the summary must remember which subset of blocks we have seen before inspecting the last block that appears after the \$. Encoding the set of blocks that were observed requires space of size $O(2^n)$. With less memory, there will be two prefixes with different sets of occurring blocks, which have the same memory representation; this means that runtime verification will not be able to check the execution correctly.

4 Monitoring *Safety* Properties

4.1 Past Propositional Temporal Logic

Safety properties are a subset of the (future) LTL properties. One can apply a decision procedure [33] to check whether an LTL property forms a safety property. However, there is an alternative way of expressing LTL safety properties, which guarantees syntactically that the given property is safety. This is based on using *past* operators for LTL, symmetric to the future operators. Let P be a finite set of *propositions*. The syntax of *past-time propositional linear time temporal logic* PLTL is defined as follows.

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

where $p \in P$.

The operator $\ominus$ (for *previous-time*) is the past mirror of the $\bigcirc$ operator, $\diamondminus$ is the past mirror of $\Diamond$, $\boxminus$ is the past mirror of $\Box$ and $\mathcal{S}$ (for *Since*) is the past mirror of $\mathcal{U}$. We can use the following additional operators: $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondminus\, \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$, $\boxminus\, \varphi = \neg \diamondminus \neg\varphi$.

Let $\sigma = e_1 \ldots e_n$ be a finite sequence of events, consisting each of a subset of the propositions P. We denote the

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

- $\sigma, i \models \mathit{true}$.
- $\sigma, i \models p$ 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 $j < k \leq i$, $\sigma, k \models \varphi$.

We can combine the past and future definitions of LTL. However, adding the past operators does not increase the expressive power of (future) LTL [16]. Based on the combined logic, we define four extensions of PLTL, which consist of a past property prefixed with one or two future operators from $\{\Diamond, \Box\}$. All extensions are interpreted over infinite sequences:

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

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

Manna and Pnueli [26] identified the LTL *safety* properties with $\Box$PLTL and the *guarantee* properties with $\Diamond$PLTL. They have also called the properties of $\Box\Diamond$PLTL and

$\Diamond\Box$PLTL *recurrence* and *obligation*, respectively. The entire set of LTL properties can be expressed as Boolean combination of recurrence and obligation properties. Except for safety and liveness properties, the Manna and Pnueli classification is orthogonal to the one that we explored here.

4.2 RV for Propositional Past Time LTL

Past properties play an important role in RV. Runtime verification of temporal specifications concentrates in many cases on the class of properties $\Box$PLTL. Further, instead of checking a property of the form $\Box\varphi$, one often checks whether φ holds for the trace observed so far, returning *true/false* output. When the current trace violates φ, then the fact that $\Box\varphi$ fails can be concluded.

The RV algorithm for past LTL, presented in [20] 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 $i-1$. This becomes clearer when 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 both $(\sigma,i) \models \varphi$ and $(\sigma,i-1) \models (\varphi\mathcal{S}\psi)$.

The semantic definition of past LTL 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 monitors a trace against a past temporal property η. It uses two vectors of values indexed by subformulas: pre, which summarizes the truth values of the subformulas for the observed prefix *without* its last event and now, for the observed prefix *including* its last event.

1. Initially, for each subformula φ of η, $\mathsf{now}(\varphi) := \mathit{false}$.
2. Observe a new event e (as a set of propositions) 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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(p) := p \in e$.
   - $\mathsf{now}(\mathit{true}) := \mathit{true}$.
   - $\mathsf{now}((\varphi\wedge\psi)) := \mathsf{now}(\varphi)$ *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.

As opposed to the monitoring algorithm for future LTL, presented in Section 3.2, which uses a summary exponential in the size of the monitored property, this algorithm has a summary and an incremental complexity that is linear in the length of the specification.

4.3 From Monitoring Past Property φ to Monitoring $\Box\varphi$

We present an algorithm for specifications of the form $\Box$PLTL. Recall the algorithm for future propositional LTL presented in Section 3.2. It identifies minimal good and bad prefixes hence provides a *success/failed* verdicts for the monitored input trace with respect to the specification.

For the case of $\Box$PLTL, a simpler construction can be used. A single *deterministic* automaton $\mathcal{D}_{\Box\varphi}$ can be constructed for the specification $\Box\varphi$. Each state s of this automaton corresponds to the set of subformulas $sf(s)$ of φ that hold after a trace that is consistent with the inputs on any path that leads from the initial state to s. Calculating the transition relation is similar to updating the summary in the RV algorithm for past propositional LTL, as shown at the beginning of Section 4.2, using the two vectors pre and now. Let $s \overset{Q}{\to} s'$, where Q is the currently inspected set of propositions. Define pre as follows: for a subformula η of the given specification, $\mathsf{pre}(\eta) = \mathit{true}$ iff $\eta \in sf(s)$. For a propositional letter p, set $\mathsf{now}(p) = \mathit{true}$ iff $p \in Q$. Now, for the subformulas in $sub(\varphi)$ that do not consist solely of a proposition, calculate now as in Step 4 in the algorithm in Section 4.2. Then, for $\eta \in sub(\varphi)$, $\eta \in sf(s')$ iff $\mathsf{now}(\eta) = \mathit{true}$. The initial state consists of the empty set of subformulas. A state of $\mathcal{D}_{\Box\varphi}$ is *accepting* if it contains the formula φ itself.

One can use $\mathcal{D}_{\Box\varphi}$ to decide on verdicts when monitoring against the specification $\Box\varphi$. An (infinite) execution satisfies $\Box\varphi$ if it runs only through accepting states. Note that the number of states is $O(2^{|\varphi|})$, but each state can be represented using space linear of $|\varphi|$. We mark the states from which all the future successors are accepting by $\top$. Initially, mark every accepting node by $\top$. Then, repeatedly remove the $\top$ marking from nodes that have a successor that is *not* marked by $\top$. Keep doing that until there is no $\top$ marking that can be removed[2]. We mark states where there are no infinite accepting continuations $\bot$. To do that, start by marking the non accepting nodes by $\bot$. Repeatedly, mark by $\bot$ nodes whose entire set of successors are already marked by $\bot$. Keep doing this until no new node can be marked[3]. Finally, nodes that are not marked by $\top$ or $\bot$ are marked by ?.

Marking the states of the automaton $\mathcal{D}_{\Box\varphi}$ is done as a preliminary step. Runtime verification uses the marked automaton $\mathcal{D}_{\Box\varphi}$ to monitor input traces and return the corresponding verdict. The RV algorithm then needs to keep the current state, and can figure out the successor state based on the two-vector update. Consequently, the size of the summary, and the incremental complexity are linear in the size of φ. This can be compared to the automaton-based RV algorithm for future LTL from Section 3.2, which needs to keep a set of states of the constructed automata, hence requiring exponential space and time for each update.

We need to take the fact that the algorithm for $\Box$PLTL has a linear incremental complexity and a linear summary in the size of the specification with a grain of salt. The example in Section 3.3 shows that the summary for the presented property needs to be exponential in n. This (safety) property is presented in future LTL with a formula whose size is quadratic in n. But for $\Box\varphi$ with a past property φ, a summary that is only linear

[2] This is similar to the model checking algorithm for the CTL property $AG\top$ [7].

[3] This is similar to the model checking algorithm for the CTL property $AF\bot$.

in $|\varphi|$ is sufficient. Unfortunately, this implies that expressing this property in the form $\Box\varphi$ requires a formula whose length is exponential in n. A similar reasoning implies that for a reversed property, where the block of length n that needs to repeat appears at the *beginning* rather at the end, the property can be written with past LTL formula of quadratic size in n, but the future LTL property needs to be of length exponential in n.

Monitoring with respect to a past property φ, rather than $\Box\varphi$, the verdict can change between *true* and *false* multiple times. However, for safety properties we may be mostly interested in finding the case where the current prefix fails to satisfy φ. When the verdict for the current prefix is *false*, we can issue a *fail* verdict for $\Box\varphi$. However, it is possible that a prefix σ satisfies φ, while $\Box\varphi$ does not hold for all extensions of this prefix to an infinite execution. Consider as an example the property $\Box(\ominus\ominus \mathit{false} \vee \ominus\ominus p)$, which becomes *false* only two events after the first event where $\neg p$ holds (the disjunct $\ominus\ominus \mathit{false}$ is used to rule out failure due to fact that the failing prefix is shorter than two events). Thus, although $\Box\varphi$ should return a *fail* verdict, performing RV on the past property φ will only reveal that two events later than on the minimal trace. Monitoring $\Box\varphi$ using $\mathcal{D}_{\Box\varphi}$ would provide the *fail* verdict at the minimal trace that cannot be extended to satisfy $\Box\varphi$.

If φ holds for some observed prefix σ but $\Box\varphi$ fails on every infinite extension of σ, then we will eventually observe an extension $\sigma.\rho$ of σ, where $|\rho|$ *depends on the size of* φ and where φ does not hold. Thus, if we do not want to use the automaton $\mathcal{D}_{\Box\varphi}$ to decide when $\Box\varphi$ already holds, but instead check φ after each new event, then there is a limit to the number of steps that we need to wait until $\neg\varphi$ will fail to hold that depends on $|\varphi|$. To see this, consider a finite trace σ where φ holds and where all the infinite extensions of σ have some finite prefix that does not satisfy φ. Let n be the number of states of $\mathcal{D}_{\Box\varphi}$, which is known to be bounded by $O(2^{|\varphi|})$ [22]. Running the deterministic automaton $\mathcal{D}_{\Box\varphi}$ on σ, we end up in some state s. Suppose, for the contradiction, that there is a path ρ from s with $|\rho| > n$, where all of its prefixes satisfy φ. Then running $\mathcal{D}_{\Box\varphi}$ on the input ρ from the state s, one must pass through at least one state of $\mathcal{D}_{\Box\varphi}$ more than once. This allows constructing ("pumping") an infinite path on $\mathcal{D}_{\Box\varphi}$, where all of its states indicate that the prefix so far satisfies φ, a contradiction.

4.4 From Monitoring Propositional to First Order Temporal Logic

Runtime verification was extended to specifications that contain data. In particular, the tools DejaVu and MonPoly[4] allow specification that is based on first-order past LTL. An *event* in this case consists of predicates with parameters, i.e., in the form $q(3)$. DejaVu algorithm is restricted to checking a first order past property φ, rather than checking $\Box\varphi$.

For first order RV, the problem of discovering that a finite trace σ cannot be extended to satisfy $\Box\varphi$, although the trace itself still satisfies φ intensifies: in some cases, the maximal number of events that are required to extend σ depends to the trace σ itself, and is not a function of the specification φ. Consider the following specification φ:

$$\forall x((q(x) \rightarrow \neg\ominus \Diamond\!\!\!\!\cdot\; q(0) \vee q(x)) \wedge (r(x) \rightarrow (\ominus \Diamond\!\!\!\!\cdot\; q(x) \wedge \neg\ominus \Diamond\!\!\!\!\cdot\; r(x))))$$

[4] MonPoly allows a limited use of *finite* future, but the monitoring is then actually resolved when that future is reached.

This property asserts that events of the form $q(x)$ or $r(x)$ can appear only once with the same parameter each. Further, an $r(x)$ event can occur only if a $q(x)$ event happened with the same parameter x. Moreover, after a $q(0)$ event, no event of the form $q(x)$ can happen. Consequently, if the event $q(0)$ has happened, no further $q(x)$ events can occur, and the only events that can occur are of the form $r(x)$, where $q(x)$ has already occurred (with the same value x). Therefore, the maximal number of events that can extend the trace until φ becomes *false* is the number of $q(x)$ events that occurred for which a matching $r(x)$ event has not happened yet. Thus, once the event $q(0)$ has occurred, the verdict of $\Box\varphi$ is *fail*, although φ may still be calculated to *true* for a long while.

In [28], an algorithm that calculates the possible values of φ for extensions up to a given fixed size k is presented. However, it is shown, by a reduction from the Post Correspondence Problem decision problem, that checking $\Box\varphi$ for a first order past property φ is undecidable.

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. E. M. Clarke, E. A. Emerson: Design and synthesis of synchronization skeletons using branching-time temporal logic. Logic of Programs 1981: 52-71.
8. E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT Press, 2000.
9. D. Tabakov, K. Y. Rozier, Mo. Y. Vardi. Optimized temporal monitors for SystemC. Formal Methods in System Design 41(3): 236-268 (2012).
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. Dov M. Gabbay, Amir Pnueli, Saharon Shelah, Jonathan Stavi, On the Temporal Analysis of Fairness. POPL 1980, 163-173.
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, D. Peled, D. Ulus, First-order Temporal Logic Monitoring with BDDs. FM-CAD 2017, 116-123.
19. 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.
20. K. Havelund, G. Rosu, Synthesizing monitors for safety properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
21. H. Kallwies, M. Leucker, C. Sánchez, T. Scheffel (2022), Anticipatory Recurrent Monitoring with Uncertainty and Assumptions. Proc. of the 22nd Int'l Conference on Runtime Verification (RV'22), vol 13498 of LNCS, pp. 181-199. Springer, 2022.
22. Y. Kesten, Z. Manna, H. McGuire, A. Pnueli, A Decision Algorithm for Full Propositional Temporal Logic. CAV 1993: 97-109.
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. Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification. Springer, 1992.
27. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, STTT, Springer, 249-289, 2011.
28. M. Omer, D. Peled, Runtime Verification Prediction for Traces with Data, RV 2023, Springer Verlang, Thessaloniki, 2023.
29. D. Peled, K. Havelund, Refining the safety-liveness classification of temporal properties according to monitorability. Models, Mindsets, Meta 2018: 218-234.
30. A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers. FM'06, LNCS Volume 4085, Springer, 573-586, 2006.
31. 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.
32. S. Safra, On the complexity of omega-automata, FOCS 1988, 319-327.
33. A. P. Sistla, Safety, liveness and fairness in temporal logic, Formal Aspects of Computing 6(5): 495-512, 1994.
34. A. P. Sistla, E. M. Clarke, The complexity of propositional linear temporal logics, STOC 1982: 159-168.
35. W. Thomas, Automata on infinite objects, handbook of theoretical computer science, Volume B: Formal Models and Semantics, 133-192, 1990.
36. M. Y. Vardi, P. Wolper, Automata-theoretic techniques for modal logics of programs, Journal of Computer System Science 32(2): 183-221, 1986.