Science of Computer Programming 240 (2025) 103220

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Towards partial monitoring: Never too early to give in

Angelo Ferrando [a,*], Rafael C. Cardoso [b]

[a] *University of Modena and Reggio Emilia, Modena, 41100, Italy*
[b] *University of Aberdeen, Aberdeen, AB24 3UE, United Kingdom*

ARTICLE INFO

Keywords:
Runtime verification
Linear temporal logic
Monitorability
Partial monitoring

ABSTRACT

Runtime Verification is a lightweight formal verification technique used to verify whether a system behaves as expected at runtime. Expected behaviour is typically formally specified using properties, which are used to automatically synthesise monitors. Properties that can be verified at runtime by a monitor are called *monitorable*, while those that cannot are termed *non-monitorable*. In this paper, we revisit the notion of monitorability and demonstrate how *non-monitorable* properties can still be used to generate *partial* monitors. We tackle this from two different perspectives: (i) by recognising that a monitor can give up on monitoring the property under analysis if it recognises that the monitoring will never conclude the satisfaction or violation of the property; (ii) by recognising that a monitor can give up on events that are not necessary for successful monitoring of the property under analysis. By considering these two aspects, we present how to achieve partial monitoring of Linear Temporal Logic properties by building upon the standard monitor construction. Finally, we present a prototype implementation of our approach and its application to a remote inspection case study, as well as a set of evaluation experiments to stress test our approach using synthetic properties.

1. Introduction

Runtime Verification (RV) is a well-known lightweight formal verification technique [1]. Similar to other formal verification techniques, such as Model Checking [2] and Theorem Proving [3], it aims to verify the behaviour of the System Under Analysis (SUA), which can consist of both software and hardware components. RV achieves this verification through monitoring. Starting from a formal property expressed in a chosen formalism, one or more monitors are generated. A monitor is a device that, given a sequence of events (a trace) generated by system execution, verifies the conformance of the trace with the formal property. Since the trace can be generated at runtime, the monitor can inform system users about unexpected behaviours that violate the formal specification.

Unlike other formal verification techniques, RV is performed on the execution of the system, verifying formal properties on traces of events generated by actual system executions. This is a significant difference from traditional formal verification techniques like Model Checking, where verification is performed statically over an abstracted model of the system. RV does not require any model or additional information apart from execution traces, making it well-suited for use in “black-box” scenarios where little is known about the SUA, such as in autonomous systems. Moreover, RV performs better computationally than traditional verification techniques, as monitors only take as input what the SUA produces, without the need for a model. It has been shown that RV offers polynomial time behaviour with respect to the length of the analysed trace [4].

* Corresponding author.
E-mail addresses: angelo.ferrando@unimore.it (A. Ferrando), rafael.cardoso@abdn.ac.uk (R.C. Cardoso).

https://doi.org/10.1016/j.scico.2024.103220
Received 4 June 2024; Received in revised form 11 September 2024; Accepted 9 October 2024

Available online 17 October 2024

However, providing certification for reliable autonomous systems is challenging [5,6]. Formal verification of monolithic systems is already difficult, and applying such techniques in the context of autonomous, cyber-physical, or robotic systems is even more complicated, especially when Machine Learning techniques such as Neural Networks are involved [7]. In these scenarios, RV can be helpful, especially since it does not require a model of the system and can be deployed at runtime while the system is running. By adding monitors to the system, it is possible to improve its reliability by detecting and reacting to unexpected behaviours.

Unfortunately, some formal properties cannot be monitored at runtime. A formal property is considered *monitorable* if a monitor can be synthesised to verify it, otherwise, it is *non-monitorable*. There are various definitions in the literature regarding the requirements for a property to be considered monitorable [8]. However, the most common requirement is that a property should always allow the monitor to conclude its satisfaction or violation. RV approaches typically focus on monitorable fragments of properties for analysis. Nevertheless, as demonstrated in this paper, some non-monitorable properties may still be worth analysing at runtime, albeit partially. This paper is a revision and extension of our work in [9], where we formulated the following research question:

Is it possible for monitors synthesised from non-monitorable properties to be used in practice?

We argue that a viable answer to this question is using *partial monitoring*. Since a non-monitorable property does not assure satisfaction or violation, monitors need to be capable of *giving up* on the verification process when it is clear that no conclusion will be reached. This is especially relevant in autonomous systems, where limited computational power and memory may be available, making it essential to reclaim otherwise used resources safely. By using partial monitors, we can be less restrictive on the kind of formal properties we are allowed to use, as a non-monitorable property can still be partially monitored at runtime.

Another aspect to consider when monitoring a system is the kind of events a monitor needs to have access to. We extend our prior work by proposing a follow-up research question:

Is it possible for a monitor to give up on events as soon as they are not necessary for verifying the property under analysis?

This research question is not related to the notion of monitorability of the property under analysis. However, it is related to the notion of partial monitorability, as recognising which events are necessary for verifying the property allows the monitor to optimise resource usage at runtime. If an event is not needed, it should not be gathered and sent to the monitor to reduce the overhead introduced by the monitor in the running system.

As a proof of concept, we exemplify our approach in a robotic application where an autonomous rover is deployed into a nuclear facility for remote inspection. In this scenario, the dynamic environment makes it hard to formally verify properties using traditional methods such as model checking. Thus, we can use RV to formally verify how the rover behaves at runtime. Using this application, we show that non-monitorable formal properties have parts that can still be verified using a partial monitor, provided the monitor can detect when to give up and which events to listen to. For example, partial monitors can detect that the rover does not stay in areas with high radiation levels, but if they observe an event that would render the monitor useless, they need to give up on monitoring that property.

In this paper, we briefly revisit the notion of *monitorability*, focusing more on its engineering implications for monitor synthesis. To do so, we present a straightforward extension of the standard monitor synthesis for Linear Temporal Logic (LTL) properties, where we consider that a monitor could fail to completely verify an LTL property. We show how this can be achieved at the monitor level, instead of at the property specification level. Specifically, we reduce the problem of a monitor recognising when to give up on a property to a reachability problem inside the monitor representation.

This paper contains revisions to the content originally presented in [9], including a new algorithm for how monitors can give up on properties, and the new idea of giving up on events that are deemed to no longer be necessary. A brief complexity analysis is performed on the proposed algorithms. We have also described an entire new set of experiments by generating synthetic properties and using them to stress test our implementation and to explore what types of properties are more amenable to result in performance gains by dropping events that are no longer relevant to the properties being monitored.

The remainder of this paper is structured as follows. Section 2 presents background definitions and notation used throughout the paper. Section 3 revisits the notion of monitorability and reviews related works in the literature. Section 4 introduces our contribution, the notion of partial monitoring. Section 5 demonstrates the use of partial monitors in an autonomous rover performing remote inspection tasks. In Section 6, we provide details on how the approach described in this paper has been implemented and experimented with. Section 7 discusses the approach and its engineering implications in monitor synthesis. Finally, Section 8 concludes the paper with final remarks and future research directions.

2. Background and notation

A system S has an *alphabet* Σ containing all of its observable events. Given an alphabet Σ, a *trace* $\sigma = ev_0 ev_1 \ldots$, is a sequence of events in Σ. $\sigma(i)$ is the i-th element of σ (*i.e.*, ev_i), σ^i is the suffix of σ starting from i (*i.e.*, $ev_i\ ev_{i+1} \ldots$), Σ^* is the *set of all possible finite traces* over Σ, and Σ^ω is the *set of all possible infinite traces* over Σ.

One of the standard formalisms to specify formal properties in RV is propositional Linear Temporal Logic (LTL [10]). For this paper, the relevant parts of the syntax of LTL are as follows:

$$\varphi = true \mid false \mid ev \mid (\varphi \wedge \varphi') \mid (\varphi \vee \varphi') \mid \neg\varphi \mid (\varphi\ \mathbf{U}\ \varphi') \mid \circ\varphi$$

where $ev \in \Sigma$ is an event (a proposition), φ is a formula, $\mathbf{U}$ stands for *until*, and $\circ$ stands for *next-time*. In the rest of the paper, we also use the standard derived operators, such as $(\varphi \rightarrow \varphi')$ instead of $(\neg\varphi \vee \varphi')$, $\varphi\, R\, \varphi'$ instead of $\neg(\neg\varphi\, \mathbf{U}\, \neg\varphi')$, $\Box\varphi$ (*always* φ) instead of $(false\; R\; \varphi)$, and $\Diamond\varphi$ (*eventually* φ) instead of $(true\, \mathbf{U}\, \varphi)$.

Let $\sigma \in \Sigma^{\omega}$ be an infinite sequence of events over Σ, the semantics of LTL is as follows:

$\sigma \models ev$ if $ev = \sigma(0)$

$\sigma \models \neg\varphi$ if $\sigma \not\models \varphi$

$\sigma \models \varphi \wedge \varphi'$ if $\sigma \models \varphi$ and $\sigma \models \varphi'$

$\sigma \models \varphi \vee \varphi'$ if $\sigma \models \varphi$ or $\sigma \models \varphi'$

$\sigma \models \circ\varphi$ if $\sigma^{1} \models \varphi$

$\sigma \models \varphi\, \mathbf{U}\, \varphi'$ if $\exists_{i \geq 0} . \sigma^{i} \models \varphi'$ and $\forall_{0 \leq j < i} . \sigma^{j} \models \varphi$

In other words, a trace σ satisfies an atomic proposition (ev), if the event ev belongs to the head (first element) of σ; which means, ev has been observed as initial event of the trace σ. A trace σ satisfies the negation of the LTL property φ, if σ does not satisfy φ. A trace σ satisfies the conjunction of two LTL properties, if σ satisfies both properties. A trace σ satisfies the disjunction of two LTL properties, if σ satisfies at least one of them. A trace σ satisfies next-time φ, if the suffix of σ starting in the next step (σ^{1}) satisfies φ. Finally, a trace σ satisfies $\varphi\, \mathbf{U}\, \varphi'$, if there exists a suffix of σ s.t. φ' is satisfied, and for all suffixes before it, φ holds.

Thus, given an LTL property φ, we denote $[\![\varphi]\!]$ the language of the property, *i.e.*, the set of traces which satisfy φ; namely $[\![\varphi]\!] = \{\sigma \mid \sigma \models \varphi\}$.

In Definition 1, we present a general and formalism-agnostic definition of a runtime monitor. As mentioned before, a monitor is a function that, given a trace of events in input, returns a verdict which denotes the satisfaction (resp. violation) of a formal property over the trace.

Definition 1 *(Monitor).* Let S be a system with alphabet Σ, and φ be an LTL property. Then, a monitor for φ is a function $Mon_{\varphi,\Sigma} : \Sigma^{*} \rightarrow \mathbb{B}_3$, where $\mathbb{B}_3 = \{\top, \bot, ?\}$:

$$Mon_{\varphi,\Sigma}(\sigma) = \begin{cases} \top & \forall_{u \in \Sigma^{\omega}} . \sigma \bullet u \in [\![\varphi]\!] \\ \bot & \forall_{u \in \Sigma^{\omega}} . \sigma \bullet u \notin [\![\varphi]\!] \\ ? & otherwise \end{cases}$$

where $\bullet$ is the standard trace concatenation operator.

Intuitively, a monitor returns $\top$ if all continuations (u) of σ satisfy φ; $\bot$ if all possible continuations of σ violate φ; ? otherwise. The first two outcomes are standard representations of satisfaction and violation of a property, while the third is specific to RV. In more detail, it denotes when the monitor cannot conclude any verdict yet. This is closely related to the fact that RV can be applied to a system that is still running, and thus not all information about it is available. For instance, a property might be currently satisfied (resp. violated) by the system, but violated (resp. satisfied) in the (still unknown) future. The monitor can only safely conclude any of the two final verdicts, $\top$ or $\bot$, if it is sure such verdict will never change. The addition of the third outcome symbol ? helps the monitor to represent its position of uncertainty w.r.t. the current system execution.

A monitor function is usually implemented as a Finite State Machine (FSM), specifically a Moore machine [11,12], a FSM where the output value of a state is only determined by the state. A Moore machine can be defined as a tuple $\langle Q, q_0, \Sigma, O, \delta, \gamma\rangle$, where Q is a finite set of states, q_0 is the initial state, Σ is the input alphabet, O is the output alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function mapping a state and an event to the next state, and $\gamma : Q \rightarrow O$ is the function mapping a state to the output alphabet.

In [12], Bauer et al. present the sequence of steps to generate the corresponding Moore machine from an LTL formula φ, instantiating the $Mon_{\varphi,\Sigma}$ function, as summarised in Fig. 1.

Given an LTL property φ, a series of transformations is performed on φ, and its negation $\neg\varphi$. Considering φ in step *(i)*, first, a corresponding Büchi Automaton A^{φ} is generated in step *(ii)*. This can be obtained using Gerth et al.'s algorithm [13]. Such automaton recognises the set of infinite traces that satisfy φ (according to LTL semantics). Then, each state of A^{φ} is evaluated; the states that when selected as initial states in A^{φ} do not generate the empty language are then added to the F^{φ} set in step *(iii)*. With such a set, a

$$\begin{tikzcd}
Input & (i) Formula & (ii) NBA & (iii)\text{Emptiness per state} & (iv) NFA & (v) DFA & (vi)\text{Moore machine} \\
 & \varphi \arrow[r] & A^{\varphi} \arrow[r] & F^{\varphi} \arrow[r] & \hat{A}^{\varphi} \arrow[r] & \tilde{A}^{\varphi} \arrow[dr] & \\
\varphi \arrow[ur] \arrow[dr] & & & & & & Mon_{\varphi,\Sigma} \\
 & \neg\varphi \arrow[r] & A^{\neg\varphi} \arrow[r] & F^{\neg\varphi} \arrow[r] & \hat{A}^{\neg\varphi} \arrow[r] & \tilde{A}^{\neg\varphi} \arrow[ur] &
\end{tikzcd}$$

Fig. 1. Steps required to generate an FSM from an LTL formula φ. NBA is Non-deterministic Büchi Automaton, NFA is Non-deterministic Finite Automaton, and DFA is Deterministic Finite Automaton.

Non-deterministic Finite State Automaton $\hat{A}^{\varphi}$ is obtained from A^{φ} by simply substituting the final states of A^{φ} with F^{φ} in step *(iv)*. $\hat{A}^{\varphi}$ recognises the finite traces (prefixes) that has at least one infinite continuation satisfying φ (since the prefix reaches a state in F^{φ}). After that, $\hat{A}^{\varphi}$ is transformed (Rabin–Scott powerset construction [14]) into its equivalent deterministic version $\tilde{A}^{\varphi}$ in step *(v)*; this is possible since deterministic and non-deterministic automata have the same expressive power. The exact same steps are performed on $\neg\varphi$, which bring to the generation of the $\tilde{A}^{\neg\varphi}$ counterpart. The difference between $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ is that the former recognises finite traces which have continuations satisfying φ, while the latter recognises finite traces which have continuations violating φ. Finally, a Moore machine monitor can be generated as a standard automata product between $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ in the final step *(vi)*, where the states are denoted as tuples (q, q'), with q and q' belonging to $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$, respectively. The outputs are then determined as: $\top$ if q' does not belong to the final states of $\tilde{A}^{\neg\varphi}$, $\bot$ if q does not belong to the final states of $\tilde{A}^{\varphi}$, and ? otherwise.

Example 1. Let S be a system with alphabet $\Sigma = \{ev_1, ev_2, ev_3\}$, and $\varphi = \Diamond ev_1$ be an LTL property to verify. In natural language, φ reads as: "eventually event ev_1 is going to be observed". The Moore machine implementing the monitor function $Mon_{\varphi,\Sigma}$ is reported in Fig. 2. As long as events ev_2 and ev_3 are observed, the Moore machine will stay in the initial state with output ?. As long as it stays in this state, there might be continuations where ev_1 will never be observed (*i.e.*, the corresponding states in $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ are both finals). But, when ev_1 is observed, then the state changes to a positive state, with output $\top$. In fact, after observing ev_1, any trace determines the satisfaction of φ since there is no continuation capable of violating φ (*i.e.*, the corresponding state in $\tilde{A}^{\neg\varphi}$ is not final).

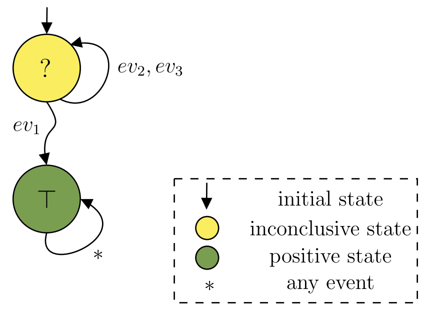

Fig. 2. Moore machine instantiated for the monitor generated by φ of Example 1.

3. Monitorability

Monitorability [8] refers to the branch of RV focused on the delineation of which formal properties can be monitored. It is crucial to understand monitorability for performing efficient verification of formal properties at runtime. However, the level of detail such notion is defined in the literature varies considerably. It has a wide range of definitions, some are more restrictive, while others are more flexible. A thorough presentation of the existing variations of concepts for monitorability can be found in [15,16], where the authors report a complete guide on monitorability and its uses.

In this paper, we consider the definitions of monitorability introduced by Pnueli and Zaks [17], where the concept of monitorability was generalised w.r.t. its first appearance [8]. We chose their view of monitorability since it is one of the most commonly cited by the community and it is less restrictive on the set of non-monitorable properties than other definitions found in the literature.

Definition 2. A property φ is σ-*monitorable*, where $\sigma \in \Sigma^*$, if there is some $u \in \Sigma^*$ such that φ is positively or negatively determined by $\sigma \bullet u$.

Definition 2 states that a property φ is considered *monitorable* with respect to a finite trace of events σ, if we can find at least one continuation trace u, such that φ is satisfied (*i.e.*, σ is a *good* prefix) or violated (*i.e.*, σ is a *bad* prefix) by the resulting concatenated trace $\sigma \bullet u$. Intuitively, if a property is σ-*monitorable*, we know that for at least one possible trace of events the monitor will be able to conclude the satisfaction or violation of φ.

Following this reasoning, we define four different notions of monitorable property. We start from less restrictive and move towards more restrictive notions. Definition 3 uses a more relaxed notion of monitorability, where a property φ is considered *existentially* monitorable when for at least one trace of events $\sigma \in \Sigma^*$ it is possible to find a continuation for which φ is either satisfied or violated. Intuitively, this means that some trace can bring the monitor to never conclude the satisfaction or violation of φ. In the literature, these properties are also known as *weak monitorable* [18].

Definition 3. A property φ is (existentially Pnueli-Zaks) $\exists_{PZ}$-*monitorable* if σ-*monitorable* for some finite trace $\sigma \in \Sigma^*$. The class of all $\exists_{PZ}$-*monitorable* properties is denoted as $\exists_{PZ}$.

Example 2. Let us assume $\varphi = (ev_1 \wedge \Diamond ev_2) \vee (ev_3 \wedge \Box\Diamond ev_4)$, and $\Sigma = \{ev_1, ev_2, ev_3, ev_4\}$. This is an example of a $\exists_{PZ}$-*monitorable* property, since we can find some $\sigma \in \Sigma^*$ for which φ is σ-*monitorable*. For example, any trace starting with ev_1 can eventually satisfy φ by observing ev_2. Furthermore, every trace $\sigma \in \Sigma^*$ starting with ev_3 is not σ-*monitorable*, since there is no continuation $u \in \Sigma^*$ s.t. $\sigma \bullet u$ positively or negatively determines φ. Fig. 3 reports the monitor obtained by φ.

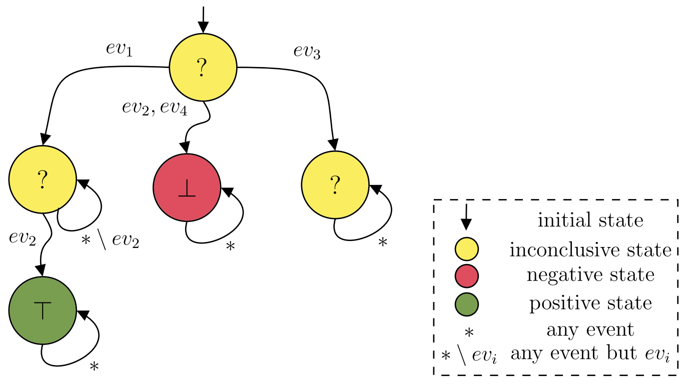

Fig. 3. Moore machine of the $\exists_{PZ}$-*monitorable* property φ presented in Example 2.

Definition 4 introduces a more restrictive notion of monitorability, where a property φ is considered *universally* monitorable when for all finite traces $\sigma \in \Sigma^*$ it is possible to find a continuation for which φ is either satisfied or violated. This means that no trace can bring the monitor to never conclude the satisfaction or violation of φ.

Definition 4. A property φ is (universally Pnueli-Zaks) $\forall_{PZ}$-*monitorable* if it is σ-*monitorable* for all finite trace $\sigma \in \Sigma^*$. The class of all $\forall_{PZ}$-*monitorable* properties is denoted as $\forall_{PZ}$.

Example 3. Let us assume $\varphi = (ev_1 \rightarrow \Diamond ev_2) \vee (ev_3 \rightarrow \Box ev_4)$, and $\Sigma = \{ev_1, ev_2, ev_3, ev_4\}$. This is an example of a $\forall_{PZ}$-*monitorable* property, since for every $\sigma \in \Sigma^*$, the property is σ-*monitorable*. We can always find a continuation $u \in \Sigma^*$ s.t. the φ is positively or negatively determined. This can be seen on the left branch where we have the possibility to satisfy the property by observing (eventually) ev_2 (positive); and on the right branch where we have the possibility to violate the property by observing something different from ev_4 (negative). Fig. 4 reports the monitor obtained by φ.

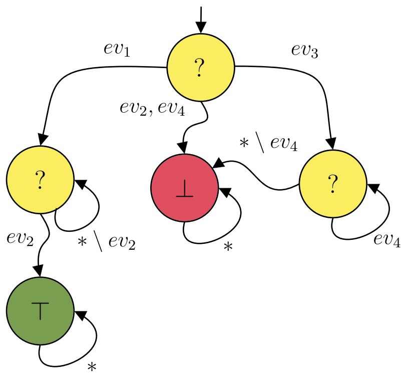

Fig. 4. Monitor (as Moore machine) of the $\forall_{PZ}$-*monitorable* property φ presented in Example 3.

Monitorable properties are defined according to Definition 4 in a variety of past research [12,1,4,19,8]. The reason for this is that any other notion of monitorability, such as the one proposed in Definition 3, does not give any guarantees on the monitor used to verify the property. Indeed, if we consider Definition 3, there is no guarantee that eventually the monitor will encounter a trace of events σ for which no continuation determines φ positively or negatively. If this happens, then the monitor will just become pointless, because it will remain in an inconclusive state forever (*i.e.*, it will never conclude anything about φ). In such scenarios, we follow the notation used in [12] to refer to such a trace of events σ as an *ugly* prefix, since it represents a case where nothing can (and nothing will) be concluded. In order to avoid these scenarios, more restrictive rules over monitorability are usually imposed; of which Definition 4 is a key example.

Next, we show that by restricting even more the notion of monitorability, we find *Safety* and *Co-Safety* properties [20]. Definition 5 denotes the properties of the kind "*nothing bad will ever happen*".

Definition 5. A property φ is a *safety* property if every $\sigma \notin [\![\varphi]\!]$ has a prefix that determines φ negatively. The class of safety properties is denoted as Safe.

These properties can only be violated at runtime, which means the resulting monitor can only report negative and inconclusive verdicts. This is due to the fact that safety properties are satisfied only by infinite traces of events, and at runtime we only have access to finite traces.

An example of a safety property can be found in the right branch of φ in Example 3, *i.e.*, $\Box ev_4$. This is a safety property where the expected behaviour is to observe ev_4 indefinitely. Thus, any $\sigma \in \Sigma^*$ (the Σ of Example 3) can be extended with a continuation $u \in \Sigma^*$ s.t. $\sigma \bullet u$ negatively determines φ. There is no continuation $u \in \Sigma^*$ s.t. $\sigma \bullet u$ positively determines φ, but this is not actually required for being monitorable.

On the same level of restrictiveness, we have *Co-Safety* properties. Definition 6 denotes the properties of the kind "something good will eventually happen".

Definition 6. A property φ is a *co-safety* property if every $\sigma \in [\![\varphi]\!]$ has a prefix that determines φ positively. The class of co-safety properties is denoted as CoSafe.

These properties can only be satisfied at runtime, which means the resulting monitor can only report positive and inconclusive verdicts. This is due to the fact that co-safety properties are violated only by infinite traces of events.

An example of a co-safety property can be found in the left branch of φ in Example 3, *i.e.*, $\Diamond ev_2$. This is a co-safety property where the expected behaviour is to observe eventually ev_2. Thus, any $\sigma \in \Sigma^*$ (the Σ of Example 3) can be extended with a continuation $u \in \Sigma^*$ s.t. $\sigma \bullet u$ positively determines φ. There is no continuation $u \in \Sigma^*$ s.t. $\sigma \bullet u$ negatively determines φ, but once again this is not required for being monitorable.

Note that, to check if a property belongs to the Safe class it is a PSPACE problem, while to check if a property is in the CoSafe class it is an EXPSPACE problem [21].

Fig. 5 represents the different monitorability classes as sets. On the left, the largest set corresponds to $\exists_{PZ}$, which only requires the properties to have at least one good prefix. Then, we have $\forall_{PZ}$, which requires the properties to have only good prefixes. After that, we find the *Safe* and *CoSafe* classes, which are included in $\forall_{PZ}$ by construction. Since for every safety (resp. co-safety) property and $\sigma \in \Sigma^*$, we may find $u \in \Sigma^*$ s.t. $\sigma \bullet u$ negatively (resp. positively) determines the property. On the right, we have the rest of the properties, which are considered *non-monitorable*. These are the properties for which there is no trace $\sigma \in \Sigma^*$ which determines the property neither positively nor negatively. Thus, properties for which all traces are ugly.

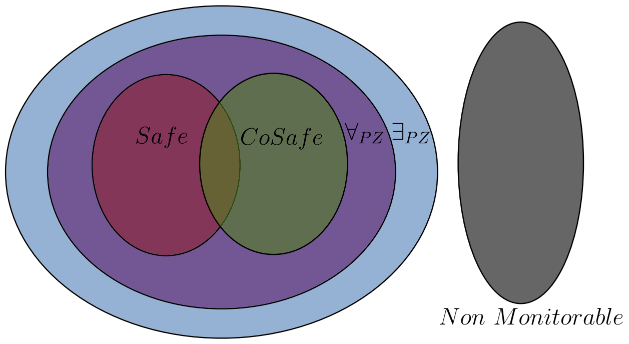

Fig. 5. Hierarchy of classes of monitorable properties.

Note that the more restrictive the conditions for a property to be considered monitorable are, the less these properties will be used in practice for achieving RV. Thus, one has to find a good balance to be able to discard as few properties as possible, and at the same time be able to correctly handle the possible lack of guarantees on the resulting monitoring process. The presence of a monitor that can reach a state with nothing to report should be avoided, or at least detected and handled.

In [17], Pnueli and Zaks propose a way to decide, given a finite trace σ, if a property under consideration φ is σ-*monitorable*. In this way, they can detect whether the current observed trace is ugly and inform the user. The main difference with our approach is at which level such check is performed. In their work this is performed on the property; while in our work we perform it directly on the monitor.

In [22,23], the notion of monitorability is presented for linear and branching flavours of Hennessy-Milner Logic with recursion (recHML) [24]. Differently from us, they consider partial monitoring the monitors which derive from either safety or co-safety properties (not both). Instead, we consider the monitors which are not always capable of determining the property, either positively or negatively.

It is important to note that a property can start monitorable (resp. non-monitorable) and become non-monitorable (resp. monitorable) depending on the information known about the SUA. In fact, as pointed out in [19], the monitorability result w.r.t. a property may change under assumptions on the SUA. If we know how the system behaves (*e.g.*, a model of the system exists), then we can rewrite the property accordingly and this can change the answer to the monitorability question.

To the best of our knowledge, our work is the first that tackles the practical implications of partial monitoring; where monitors are not assumed to be able to always conclude the satisfaction or violation of the formal property under analysis, and where it is not always simple to determine if a property is monitorable or not.

4. Partial monitoring

In this section, we introduce our two notions of partial monitoring from a practical perspective. First, we formally define how monitors can give up on properties and show an example of how they work based on a property from a previous example. Second, we describe how these monitors can drop events that are no longer relevant to them. Finally, we conclude this section by integrating both concepts into the notion of a partial monitor.

4.1. Giving up on properties

We start by introducing the first way a monitor can be partial, that is, with respect to the property under analysis. Such a monitor, unlike a standard one, should be capable of giving up on monitoring a property at runtime as soon as it realises it will never be able to conclude either the satisfaction or violation of the property.

In the previously defined Definition 1, we have the definition of a standard three-valued monitor. Usually, a monitor is intended to be *complete*, in the sense that a verdict is always assumed to be returned. This happens due to the presence of the inconclusive verdict (?), which is returned until the satisfaction ($\top$) or violation ($\bot$) of the property can be concluded. Nonetheless, in the standard definition, the property is assumed to be $\forall_{PZ}$*-monitorable*. Moreover, most of the time these are safety properties, since RV is usually applied in scenarios where it is used to verify that "nothing bad will ever happen". We expand that definition to now define our give-up monitor.

Definition 7 *(Give-Up Monitor).* Let S be a system with alphabet Σ, and φ be an LTL property. Then, a Give Up Monitor for φ is a function $Mon^{G-up}_{\varphi,\Sigma} : \Sigma^* \to \mathbb{B}_4$, where $\mathbb{B}_4 = \{\top, \bot, ?, \chi\}$:

$$Mon^{G-up}_{\varphi,\Sigma}(\sigma) = \begin{cases} \top & \forall_{u \in \Sigma^\omega} . \sigma \bullet u \in [\![\varphi]\!] \\ \bot & \forall_{u \in \Sigma^\omega} . \sigma \bullet u \notin [\![\varphi]\!] \\ ? & \exists_{u \in \Sigma^*} . ((\forall_{u' \in \Sigma^\omega} . \sigma \bullet u \bullet u' \in [\![\varphi]\!]) \vee \\ & \qquad (\forall_{u' \in \Sigma^\omega} . \sigma \bullet u \bullet u' \notin [\![\varphi]\!])) \\ \chi & otherwise \end{cases}$$

where $\bullet$ is the standard trace concatenation operator.

Definition 7 presents the notion of a *Give Up Monitor*, which differs from Definition 1 in the values returned as outcome of the verification. An additional output "give up" value is added, *i.e.*, χ. With χ, the monitor can explicitly give up on the current execution and inform the user/system that there is no point in continuing to monitor this property. To make the addition of this new output possible, we updated the condition for returning ?. The monitor now requires the existence of a future continuation of σ which will make the monitor conclude with a final verdict ($\top$ or $\bot$). If that is the case, then the monitor can conclude (for the moment) an inconclusive verdict, and eventually, it might conclude a final verdict. Otherwise, the monitor is unfortunately in a situation where σ denotes an ugly prefix, where no possible continuation will ever allow the monitor to conclude the satisfaction or violation of φ. When this happens the monitor returns χ, which symbolises that it has given up on the current analysis.

Algorithm 1 describes how to synthesise a *Give Up Monitor*. Initially, given an LTL property φ and an alphabet Σ as input, Algorithm 1 synthesises a standard LTL monitor according to [12] (line 1). Subsequently, the algorithm loops over the set of states of the resulting Moore machine (lines 2–13). For each state q, the algorithm checks the current outcome $\gamma(q)$ (line 3). If the outcome is equal to 'unknown' (*i.e.*, ?), the algorithm checks whether a final state q' can be reached from q; where a state q' is considered final if its outcome is either $\top$ or $\bot$ (line 6). If such a state q' can be found, then the outcome of q is left unchanged (line 7). Otherwise, it means that no final state q' can be reached from q according to the δ transition function and the outcome associated with q is changed to 'give up' (*i.e.*, χ). This is achieved by setting the auxiliary variable o to χ (line 4) and not changing it in the nested loop (lines 5–10).

Once the analysis has been carried out on all the states in Q, the revised monitor is returned (line 14). Note that, the resulting *Give Up Monitor* is equivalent to the standard monitor (obtained in line 1) but with an additional outcome symbol (that is the χ one) and with an updated γ function to map the states, taking into consideration the newly introduced symbol.

Theorem 1. *Algorithm 1 terminates in double exponential time with respect to the size of φ.*

Algorithm 1 *GiveUpMonitor*(φ, Σ).

```
1: $\langle Q, q_0, \Sigma, O, \delta, \gamma \rangle = Monitor(\varphi, \Sigma)$
2: for $q \in Q$ do
3:     if $\gamma(q) = ?$ then
4:         $o = \chi$
5:         for $q' \in Q$ do
6:             if $\gamma(q') \in \{\top, \bot\} \wedge q' = \delta(\delta(\delta(q, ev_0), ev_1), \ldots)$ then
7:                 $o = ?$
8:                 break
9:             end if
10:        end for
11:        $\gamma(q) = o$
12:    end if
13: end for
14: return $\langle Q, q_0, \Sigma, O \cup \{\chi\}, \delta, \gamma \rangle$
```

Proof. Given a standard monitor, to obtain a *Give Up Monitor* we add an additional post-processing step after generating the Moore machine (lines 2–14). From a Moore machine representing the instantiation of a monitor, we compute for each state labelled with ? the reachability of a state labelled with $\top$ or $\bot$. Reaching these states means that the monitor cannot get stuck in an inconclusive state indefinitely (*i.e.*, it has no dead ends). This analysis can be achieved in polynomial time w.r.t. the number of states and edges of the Moore machine. However, the size of the Moore machine is double exponential with respect to the size of the formula φ, thus the complexity of Algorithm 1 follows. □

Corollary 1. *The double exponential time complexity of Algorithm 1, as stated in Theorem 1, specifically applies to the case where the specification φ is expressed in LTL. However, if the specification φ is expressed in a logic with different expressive power or complexity characteristics, the time complexity of Algorithm 1 may differ. For instance, in logics where monitor generation is polynomial or single exponential with respect to the size of the specification, the overall complexity of the give-up monitoring procedure could be correspondingly reduced.*

Proof. The proof of Theorem 1 is based on the fact that the size of the Moore machine generated from an LTL formula is double exponential with respect to the size of the formula φ. This accounts for the double exponential complexity of the give-up monitoring process in this context. However, for logics where the monitor generation process is less complex—such as those where monitors can be generated in polynomial or single exponential time relative to the size of the specification—the time complexity of the give-up monitor creation would correspondingly reflect the reduced complexity of monitor generation. □

To better understand how Algorithm 1 works, let us now consider an example of use.

Example 4. Considering once again the property of Example 2, $\varphi = (ev_1 \wedge \Diamond ev_2) \vee (ev_3 \wedge \Box\Diamond ev_4)$, with $\Sigma = \{ev_1, ev_2, ev_3, ev_4\}$. This property is $\exists_{PZ}$-*monitorable*, since not all $\sigma \in \Sigma^*$ are σ-*monitorable*. For this reason this property would usually be discarded, since no guarantees can be given that the resulting monitor will be able to conclude anything. In this case, by following Definition 7, we can update the monitor with the additional outcome to represent the cases where it should give up. Fig. 6 reports the partial monitor obtained by updating the monitor from Fig. 3 according to Algorithm 1.

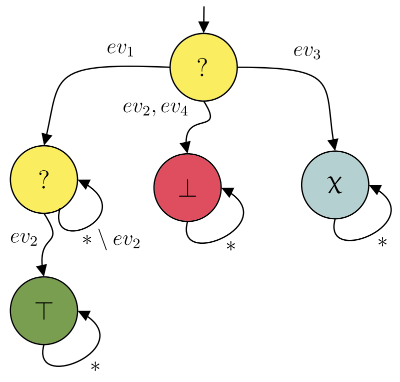

Fig. 6. *Give Up Monitor* of the $\exists_{PZ}$-*monitorable* property φ presented in Example 2. Here, we can note how the previously inconclusive state on the right has now become a “give up” state (grey colour).

4.2. Giving up on events

Besides giving up on a property, a monitor can also give up on analysing certain events at runtime. In other words, even if a property is worth being monitored, not all events in Σ may be useful. This aspect is related once again to the standard definition of a monitor, where the trace of events given in the input σ is assumed to always be complete in terms of the events in Σ. However, this completeness is not always necessary, as a property may not require the same set of events throughout the entire monitoring process. Building on this observation, we can define another aspect on which monitors can be optimised. We do so by starting from the definition of a monitor that, instead of returning a boolean verdict, returns a set of events.

Definition 8 *(Events Monitor)*. Let S be a system with alphabet Σ, and φ be an LTL property. Then, an Events Monitor for φ is a function $Mon^{ev}_{\varphi,\Sigma} : \Sigma^* \rightarrow \mathcal{P}(\Sigma)$:

$$Mon^{ev}_{\varphi,\Sigma}(\sigma) = \begin{cases} \emptyset & Mon^{G-up}_{\varphi,\Sigma}(\sigma) \in \{\top, \bot, \chi\} \\ Ev & \forall_{ev \in \Sigma}.(ev \in Ev \iff Mon^{G-up}_{\varphi,\Sigma}(\sigma) \neq Mon^{G-up}_{\varphi,\Sigma\setminus\{ev\}}(\sigma_{\setminus\{ev\}})) \end{cases}$$

where $\sigma_{\setminus\{ev\}}$ is the trace σ removed of all the occurrences of the event ev.

The monitor presented in Definition 8 denotes a monitor capable of returning the set of events necessary for proper monitoring. This means that if an event can influence the monitor's outcome, it must be included in the returned set. On the other hand, an event that has no effect on the future outcome of the monitor can, and should, be removed. This is captured by the second case in Definition 8, where the set Ev of events belonging to Σ is constructed based on the events $ev \in \Sigma$ causing $Mon^{G-up}_{\varphi,\Sigma}$ to change its outcome. Note that, in the second call to $Mon^{G-up}_{\varphi,\Sigma\setminus\{ev\}}$, the event ev is removed from both the set Σ and the trace σ; this causes the new synthesis of a monitor that now does not care about the event ev anymore.

Algorithm 2 details how to synthesise an *Events Monitor* from a *Give Up Monitor*. First, it synthesises the *Give Up Monitor* according to Algorithm 1. Then, it creates an auxiliary list Q' (line 2). Next, for each state in Q (lines 3–8), if the state is final (line 4), it adds it to Q' (line 5). After this loop, Q' contains all states of the Moore machine with outcomes $\top$ or $\bot$. Then, the algorithm initialises an auxiliary set called $visited$ (line 9) to track already visited states to handle loops in the Moore machine.

The algorithm continues by iterating over all states in Q' in order (note that in line 11, the state q' is removed from the head of the list Q'). Each state q' is added to the set $visited$, and then the auxiliary function Pre (reported in Algorithm 3) is called on Q and q', which returns the set of predecessor states q'' that can move to q according to the δ transition function. For each of these states q'', the function Pre associates the event ev consumed in the transition. That is, if $q'' \in Pre(Q, q)$, then there exists an event $ev \in \Sigma$ such that $\delta(q'', ev) = q$. Once the set $preSet$ is generated (line 13), the algorithm iterates over all states q'' denoting the predecessors of q. For each q'', if this is the first time the algorithm encounters it, the state is appended at the end of the list Q'. After that, the mapping function γ for state q'' is updated, taking into consideration the event ev (given by the Pre function) (line 18).

Algorithm 2 $EventsMonitor(\varphi, \Sigma)$.

```
1: $\langle Q, q_0, \Sigma, O, \delta, \gamma \rangle = GiveUpMonitor(\varphi, \Sigma)$
2: $Q' = [\,]$
3: for $q \in Q$ do
4:     if $\gamma(q) \neq ?$ then
5:         append $q$ in $Q'$
6:     end if
7:     $\gamma(q) = \emptyset$
8: end for
9: $visited = \emptyset$
10: while $|Q'| > 0$ do
11:     remove first $q'$ from $Q'$
12:     $visited = visited \cup \{q'\}$
13:     $preSet = Pre(Q, q')$
14:     for $\langle q'', ev \rangle \in preSet$ do
15:         if $q'' \notin visited \wedge q'' \notin Q'$ then
16:             append $q''$ in $Q'$
17:         end if
18:         $\gamma(q'') = \gamma(q'') \cup \{ev\}$
19:     end for
20: end while
21: return $\langle Q, q_0, \Sigma, \mathcal{P}(\Sigma), \delta, \gamma \rangle$
```

Once the loop is concluded (*i.e.*, all states have been visited), the algorithm returns the resulting *Events Monitor*, which corresponds to the standard monitor with the updated γ mapping function and a corresponding output alphabet (set to the powerset of Σ).

To summarise, Algorithm 2 starts from the final states of the Moore machine and backtracks to all other states that can reach such states (directly or indirectly). For each of these states, a set of events is stored containing the events necessary to reach any of the final states.

Algorithm 3 $Pre(Q, q)$.

```
1: $preSet = \emptyset$
2: for $q' \in Q \setminus \{q\}$ do
3:     for $ev \in \Sigma$ do
4:         if $\delta(q', ev) = q$ then
5:             $preSet = preSet \cup \{\langle q', ev \rangle\}$
6:         end if
7:     end for
8: end for
9: return $preSet$
```

Theorem 2. *Algorithm 2 terminates in double exponential time with respect to the size of φ.*

Proof. Algorithm 2 first synthesises a standard monitor (line 1). This process requires double exponential time with respect to the size of the LTL formula $|\varphi|$ (according to Theorem 1). After that, Algorithm 2 iterates over the states of the resulting Moore machine (lines 3–8). Then, the algorithm loops over the states of the Moore machine once more (lines 10–20), where for each state, all possible predecessor states are considered (i.e., all incoming edges to the state). Thus, the time complexity of Algorithm 2 is polynomial with respect to the size of the Moore machine, that is of double exponential size with respect to the size of φ. $\square$

Corollary 2. *Let $\mathcal{M} = \langle Q, q_0, \Sigma, O, \delta, \gamma \rangle$ be the Moore machine synthesised by Algorithm 1 for an LTL formula φ, where Q is the set of states, q_0 is the initial state, Σ is the input alphabet, $O = \{\top, \bot, ?, \chi\}$ is the set of possible outputs, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\gamma : Q \to O$ is the output function. Let $\mathcal{M}_{ev} = \langle Q, q_0, \Sigma, P(\Sigma), \delta, \gamma' \rangle$ be the monitor produced by Algorithm 2, where the output function $\gamma' : Q \to P(\Sigma)$ maps each state to a set of relevant events. Then, the transformation from γ to γ' does not alter the state transitions of the monitor $\mathcal{M}_{ev}$ but changes the output from boolean values $\{\top, \bot, ?, \chi\}$ to the set of events $P(\Sigma)$ that influence state transitions. Specifically, for any state $q \in Q$, $\gamma'(q)$ identifies the events in Σ that are necessary for transitioning from state q.*

Proof. Algorithm 2 modifies the output function γ of the Moore machine $\mathcal{M}$ by replacing the boolean verdicts $\{\top, \bot, ?, \chi\}$ with a set of events from Σ that are essential for state transitions. The transition function δ remains unchanged, ensuring that the sequence of state transitions in $\mathcal{M}_{ev}$ is identical to that in $\mathcal{M}$. The transformation thus shifts the monitor's focus from producing verdicts to identifying relevant events, optimising the monitoring process without affecting the underlying state transitions. $\square$

Let us consider an example to show how an events monitor can be useful in practice.

Example 5. Considering again the property of Example 2, $\varphi = (ev_1 \wedge \Diamond ev_2) \vee (ev_3 \wedge \Box\Diamond ev_4)$, with $\Sigma = \{ev_1, ev_2, ev_3, ev_4\}$. Since not all $\sigma \in \Sigma^*$ are *σ-monitorable*, this property does not need to use all events at runtime for its actual monitoring. Specifically, as shown in Fig. 7, we can synthesise a monitor from the one reported in Fig. 6 according to Algorithm 2. In such a monitor, each state's outcome determines the events of interest in such a state. For instance, in the initial state, all events are necessary for monitoring φ. This makes sense since all events can bring to conclusive states (so they cannot be removed). Instead, in the state reached by observing the ev_1 event in the initial state, the only event of interest is ev_2. Again, this makes sense since upon reaching this state, the only event that can make the monitor change its mind is event ev_2. This means that all the other events are useless for the monitor and can be safely removed.

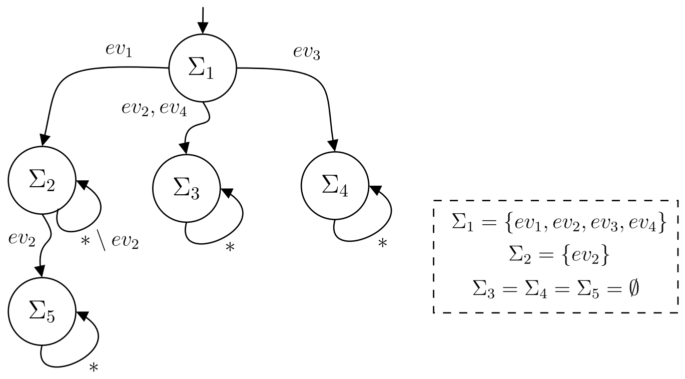

Fig. 7. *Events Monitor* of the $\exists_{PZ}$*-monitorable* property φ presented in Example 2. Here, we can note how the outcomes are not boolean values but set of events (all included in $\mathcal{P}(\Sigma)$).

4.3. Partial monitor

Now that we have presented both *Give Up* and *Events Monitors*, we can focus on their integration to build the notion of a partial monitor.

Definition 9 *(Partial Monitor)*. Let S be a system with alphabet Σ, and φ be an LTL property. Then, a Partial Monitor for φ is a function $Mon^{partial}_{\varphi,\Sigma} : \Sigma^* \to \mathbb{B}_4 \times \mathcal{P}(\Sigma)$, where $\mathbb{B}_4 = \{\top, \bot, ?, \chi\}$:

$$Mon^{partial}_{\varphi,\Sigma}(\sigma) = \langle Mon^{G-up}_{\varphi,\Sigma}(\sigma), Mon^{ev}_{\varphi,\Sigma}(\sigma)\rangle$$

where $\bullet$ is the standard trace concatenation operator.

Definition 9 presents the notion of a *Partial Monitor* as a combination of a *Give Up* and *Events Monitor*. Specifically, given an LTL formula φ and a trace of events σ, a partial monitor returns a tuple containing the boolean satisfaction for formula φ on trace σ, and the set of events that the partial monitor still considers of interest for the verification.

$$\begin{tikzcd}
Input & (i)-(v) & (vi)\text{ Moore machine} & (vii)\text{ Give Up} & (viii)\text{ Events} & (ix)\text{ Partial Monitor} \\
\varphi \arrow[r] & \ldots \arrow[r] & Mon_{\varphi,\Sigma} \arrow[r] & Mon^{G-up}_{\varphi,\Sigma} \arrow[rr] \arrow[dr] & & Mon^{partial}_{\varphi,\Sigma} \\
& & & & Mon^{ev}_{\varphi,\Sigma} \arrow[ur] &
\end{tikzcd}$$

Fig. 8. Steps required to generate a partial monitor (steps from (i) to (v) are omitted and can be found in Fig. 1).

Fig. 8 reports the updated pipeline to synthesise a partial monitor given an LTL formula φ. With respect to Fig. 1, the additional steps (vii) – (ix) are added to synthesise a partial monitor starting from the standard one, obtained in the last step of Fig. 1. Note that, the partial monitor can be obtained as the product of the two Moore machines produced by Algorithm 1 and Algorithm 2.

Corollary 3. *Let S be a system with alphabet Σ, and φ an LTL formula, the synthesis of a Partial Monitor $Mon^{partial}_{\varphi,\Sigma}$ can be achieved in double exponential time with respect to the size of φ.*

Proof. It derives directly from Theorem 1 and Theorem 2. $\square$

We now conclude by reporting the final version of our running example where we combine the results of Example 4 and Example 5 according to Definition 9.

Example 6. Considering once again the property of Example 2, $\varphi = (ev_1 \wedge \Diamond ev_2) \vee (ev_3 \wedge \Box\Diamond ev_4)$, with $\Sigma = \{ev_1, ev_2, ev_3, ev_4\}$. We can combine the Give Up Monitor obtained in Example 4 with the Events Monitor obtained in Example 5, resulting in the Moore machine shown in Fig. 9.

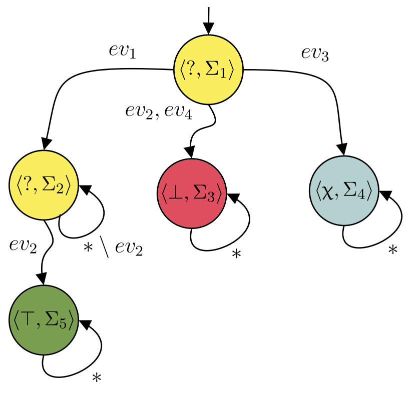
ev_1
ev_3
$\langle ?, \Sigma_1\rangle$
ev_2, ev_4
$\langle ?, \Sigma_2\rangle$
$\langle \bot, \Sigma_3\rangle$
$\langle \chi, \Sigma_4\rangle$
ev_2
$* \setminus ev_2$
$*$
$*$
$\langle \top, \Sigma_5\rangle$
$*$

Fig. 9. *Partial Monitor* of the $\exists_{PZ}$*-monitorable* property φ presented in Example 2. The Σ_i sets of events are the same of Fig. 7.

5. Remote inspection case study

This section presents a more realistic case study, highlighting how our approach can be effectively employed in different domains. Unlike the synthetic experiments presented later on in this paper, this section does not focus on performance aspects but rather on demonstrating our approach in a more practical scenario.

We demonstrate the usefulness of our approach by applying it to a remote inspection case study. This case study is based on a simulation, first introduced in [25], of an autonomous rover deployed to perform remote inspection of nuclear facilities. The rover has access to sensors which are used to detect the level of radiation, and a camera which is used to acquire images of tanks containing radioactive material to perform integrity analysis (*e.g.*, deterioration of the container). The objective of the rover is to patrol and inspect important locations (*i.e.*, marked as waypoints) around the facility. As part of an inspection task, the rover has to take measurements of the radiation level when it arrives in such locations.

Example 7. We start by demonstrating our approach applied to this example with a very simple property $\varphi = \Box\Diamond inspect\_tank\_1$. This property is shown in Fig. 10, with Fig. 10a containing the traditional Moore machine, and then after applying our technique we can see in Fig. 10b that the monitor can only give up in this case. Intuitively, this property states that it is always the case that eventually the rover will inspect the waypoint *tank1*. Since the rover has to constantly patrol these waypoints, it makes sense to represent this behaviour with such property. However, we note that there are many other ways to write this property, and some may sacrifice generalisation to write a property that is monitorable. That is a valid approach, and for simple cases such as with this property it is indeed the best solution, since after applying our approach we ended up with a monitor that is only able to give up, *i.e.*, no partial monitoring is possible, and therefore this property is non-monitorable.

(a) Traditional Moore machine monitor generated from property φ.

(b) New monitor generated after applying our approach.

Fig. 10. A simple property $\varphi = \Box\Diamond inspect\_tank\_1$.

Example 8. Next, let us consider a more interesting property where we can demonstrate that partial monitoring can indeed be useful:

$$\begin{aligned}\varphi = radiation\_low\ \mathbf{U}((&radiation\_high \wedge \Diamond move\_to\_decontamination) \vee \\ &(radiation\_medium \wedge \Box\Diamond(inspect\_tank\_1 \vee inspect\_tank\_2 \vee \\ &\ldots \vee inspect\_tank\_n)))\end{aligned}$$

This property says that we can observe the event radiation_low until we observe either radiation_high or radiation_medium. Low, medium, and high radiation refer to the level of radiation that is currently observed by the radiation sensor. If radiation_high is observed, then eventually we have to observe the event move_to_decontamination, which represents the command being sent to move the rover to a decontamination zone since a high level of radiation can be dangerous to the rover. Otherwise, if we observe radiation_medium, then we have to inspect one of the radiation tanks ($1 \ldots n$) to identify if there are any abnormalities.

The Moore machine monitor for this more complex property is shown in Fig. 11. This monitor originally has three inconclusive states, as shown in Fig. 11a. The first is the initial state which will stay inconclusive when it observes *rad_low*, until it observes *rad_high* and then moves to the left branch, or *rad_medium* and then moves to the right branch, or any other event and then moves to the centre branch. Since the initial state is inconclusive, we have to expand it to look for positive and negative states. The centre branch is immediately expanded into a negative state, thus, we know that the initial state should remain inconclusive (*i.e.*, not output give up). Looking at the left branch, we encounter the second inconclusive state, but we can quickly notice that upon observing *mv_dec* we arrive in a positive state, thus, we also know that the inconclusive state in the left branch should remain inconclusive. Finally, the third and last inconclusive state can be found in the right branch. There is no transition from this state to any other state that leads into positive or negative states, therefore, this state should output give up. Fig. 11b contains the result of applying Algorithm 1 and Algorithm 2 on the Moore machine.

Example 9. The partial monitor discussed in Example 8 illustrates how a monitor can give up on formula verification and events in the remote inspection case study. However, we acknowledge that the partial monitor shown in Fig. 11b only allows giving up on a subset of the radiation atomic propositions, not on the entire sensor itself. To better understand the impact of dropping events in a partial monitor from the remote inspection case study, let us now focus on the following LTL formula:

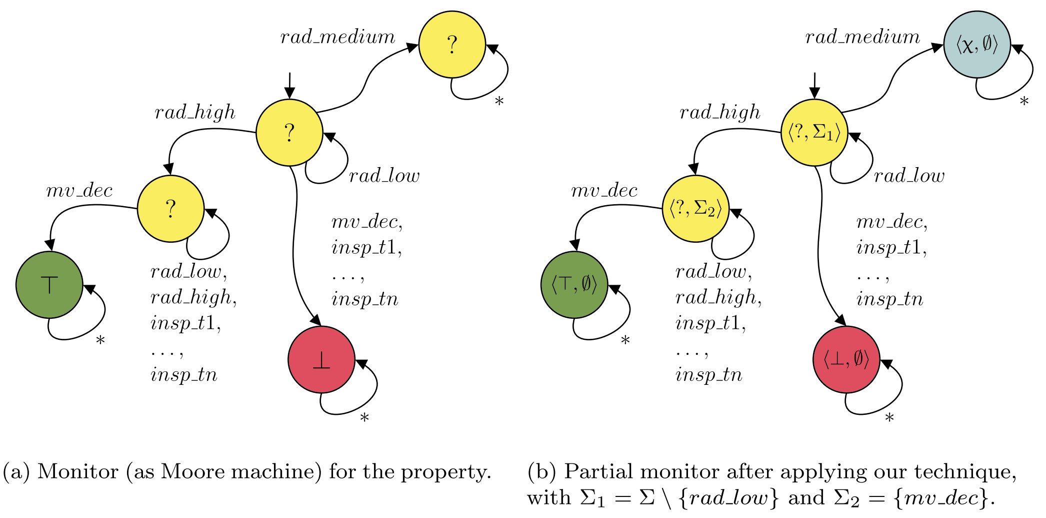
rad_medium
?
*
rad_high
?
rad_low
mv_dec
?
mv_dec, insp_t1, ..., insp_tn
⊤
rad_low, rad_high, insp_t1, ..., insp_tn
⊥
(a) Monitor (as Moore machine) for the property.
⟨χ, ∅⟩
⟨?, Σ1⟩
⟨?, Σ2⟩
⟨⊤, ∅⟩
⟨⊥, ∅⟩
(b) Partial monitor after applying our technique, with Σ1 = Σ \ {rad_low} and Σ2 = {mv_dec}.

Fig. 11. A property that deals with the different levels of radiation. *rad* is short for radiation, *insp* is short for inspection, *t* is short for tank, and *mv_dec* is short for move to decontamination.

$$\varphi = (\neg cut\_tank\_1 \wedge \ldots \wedge \neg cut\_tank\_n)\,\mathbf{U}\,\bigcirc radiation\_high \vee$$
$$\Diamond move\_to\_decontamination$$

This formula specifies that if the rover's camera detects a cut (considered an abnormality) on any inspected tank, the radiation sensor should register a high radiation level. Regardless of the radiation level detected, the rover should then return to base for decontamination.

The Moore machine monitor for formula φ from Example 9 is shown in Fig. 12. In this partial monitor, when the camera detects an abnormality in a tank (e.g., a cut), the monitor transitions to a state (bottom right) where it disregards the radiation sensor, thereby avoiding unnecessary data exchange. This is evident from Σ_2, which only includes the atomic proposition *move_to_decontamination*.

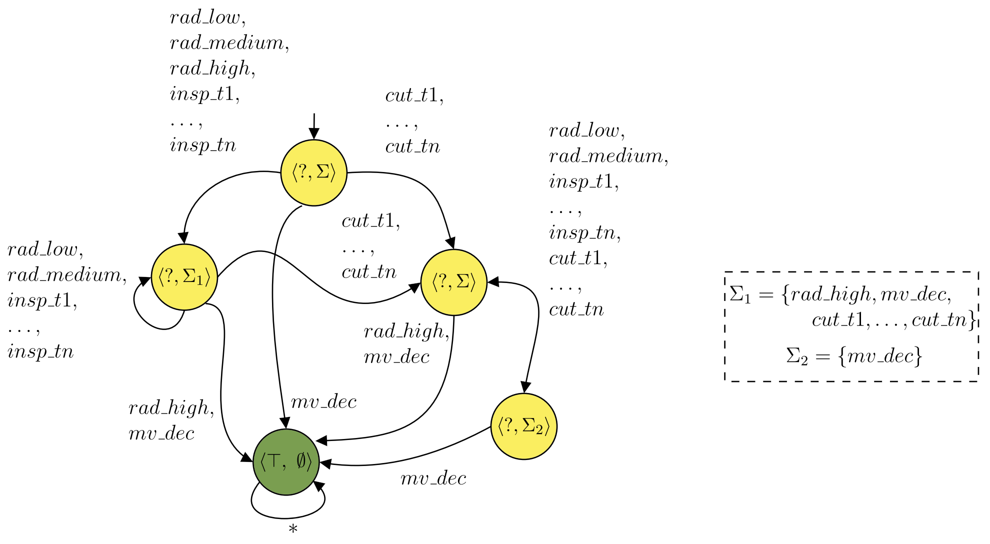
rad_low, rad_medium, rad_high, insp_t1, ..., insp_tn
cut_t1, ..., cut_tn
⟨?, Σ⟩
rad_low, rad_medium, insp_t1, ..., insp_tn, cut_t1, ..., cut_tn
rad_low, rad_medium, insp_t1, ..., insp_tn
⟨?, Σ1⟩
cut_t1, ..., cut_tn
⟨?, Σ⟩
rad_high, mv_dec
rad_high, mv_dec
mv_dec
⟨?, Σ2⟩
⟨⊤, ∅⟩
mv_dec
*
Σ1 = {rad_high, mv_dec, cut_t1, ..., cut_tn}
Σ2 = {mv_dec}

Fig. 12. Partial monitor for formula φ of Example 9, where *cut_tank_1*, …, *cut_tank_n* (abbreviated as *cut_ti* with $1 \le i \le n$) are atomic propositions added to denote tank abnormalities.

6. Implementation

We implement our tool[1] (along with a release[2]) for the transformation from standard to partial monitor using Java. This tool depends on LamaConv,[3] a Java library that is capable of translating temporal logics expressions into equivalent automata, and then to generate runtime monitors out of these automata. These monitors can be used for runtime verification and/or model checking. Our tool calls LamaConv and makes it generate a standard three-valued LTL monitor. After that, for each inconclusive state it performs a reachability analysis. Each inconclusive state that cannot reach any non-inconclusive state (*i.e.*, $\top$ or $\bot$ labelled states) is then labelled with χ instead of ?. In this way, monitors can still be generated and used for partial monitoring of non-monitorable properties, since ugly prefixes are explicitly recognised and the monitoring process consequently interrupted.

From an engineering perspective, our tool takes as input an LTL property and then provides a human-readable output. To be more precise, three input parameters have to be set when executing our tool:

1. the path to the folder containing LamaConv installation (where "*rltlconv.jar*" can be found);
2. φ, the LTL property that will be used for synthesising the monitor (standard LTL syntax as accepted by LamaConv);
3. Σ, the alphabet of the SUA.

Given the input described above, our tool starts by calling LamaConv, which generates a character string in the intuitive Automata File Format (AFF) format. This is then parsed by our tool into a Java object, and the reachability analysis is performed to detect give up states and events.

Our tool can generate two outputs, the updated AFF and/or a Java object. The AFF can be directly read and processed as a string representation of the monitor. When parsing this updated AFF into an application, the user should be aware that two changes are made. The first one that our tool makes to the AFF is to update the inconclusive states that were identified to never conclude the satisfaction or violation of the property with a give up symbol (by replacing "?" with "x" in the AFF). The second change our tool makes to the AFF is to add the set of events of interest in the outcome of each state of the monitor. As in Definition 9, the monitor outcome is split into two: the boolean verdict and the set of events needed to continue the monitoring. Otherwise, if using the Java object, then the tool can be included directly as an external Java library, and the monitor can be used as a Java object by calling the available methods. In this way, the monitor generated by the tool can be easily integrated with third-party software.

6.1. Evaluation

In this section, we present the experiments we used to evaluate our tool. These experiments are synthetic stress tests designed to evaluate the tool's performance under challenging conditions. These experiments are conducted on synthetic data, but they still provide valuable insights into the tool's behaviour and performance when processing thousands of LTL formulas.

The machine used to run the experiments features an 11th Gen Intel Core i9-11900KF, 3.50 GHz, 8 cores, 16 threads, 66 GB DDR4 RAM, running Ubuntu 22.04.2 LTS, with openjdk 17.0.10 2024-01-16.

Two types of experiments have been carried out to analyse and stress the implementation. The first type concerns the monitor synthesis time, which is the time the tool takes to synthesise a partial monitor given an LTL formula or a Moore machine as input. The second type concerns how many data a partial monitor can avoid producing to achieve verification. This involves determining how many data can be dropped and not sent to the monitor thanks to its being capable of understanding which events are of interest and which are not. The LTL formulas used in these experiments are randomly generated; specifically, the experiments have been carried out on over 10,000 randomly generated properties.[4]

We start by analysing the monitor synthesis time. Fig. 13 reports the results obtained. The x-axis represents the size of the LTL formulas under analysis (*i.e.*, the number of temporal and logical operators in the LTL formula, for instance $\Diamond p$ has size 1, while $\Box\Diamond p \wedge \bigcirc q$ has size 4), while the y-axis represents the execution time required by the tool to complete the synthesis of the monitors. For each property generated, the times required to synthesise a standard LTL monitor and a novel partial monitor are reported. We can observe that the time required to perform the post-processing steps does not significantly impact the overall synthesis time. Indeed, the partial monitor synthesis behaves very similarly to the standard monitor synthesis. This result is very positive and empirically shows that the post-processing step does not influence the synthesis time, which is mainly determined by the initial LTL transformation (which, as reported previously, grows exponentially with respect to the size of the property).

To facilitate the empirical evaluation of our approach, we generated random Moore machines. This was achieved by defining a set of Moore machines with a specified number of states, each labelled with an output value selected from $\{\top, \bot, ?\}$, corresponding to the different verdicts. An initial state was then assigned, and transitions between states were created based on a randomly selected input alphabet. For each state, transitions were defined for every possible input symbol. The generated Moore machines were formatted to be compatible with existing verification tools, ensuring seamless integration into our evaluation framework. This method enabled the creation of diverse and complex Moore machines, allowing for a thorough assessment of the performance and scalability of our techniques across various scenarios.

[1] https://github.com/AngeloFerrando/PartialMonitor Accessed on 02-June-2024.
[2] https://github.com/AngeloFerrando/PartialMonitor/releases/tag/v1.0 Accessed on 02-June-2024.
[3] https://www.isp.uni-luebeck.de/lamaconv Accessed on 02-June-2024.
[4] Using the Spot library (https://spot.lre.epita.fr/) Accessed on 02-June-2024.

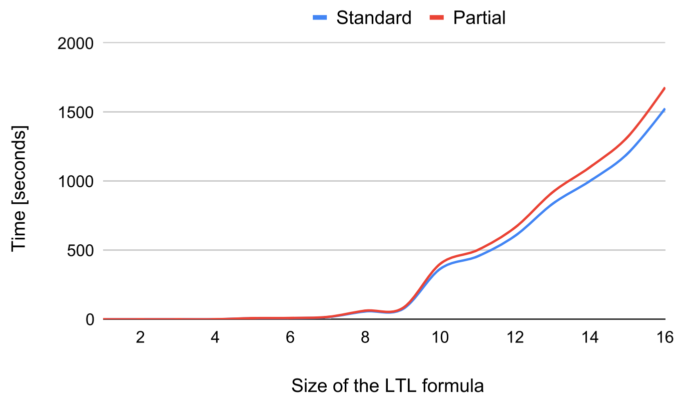

Fig. 13. Monitor synthesis time for the standard and the partial monitor.

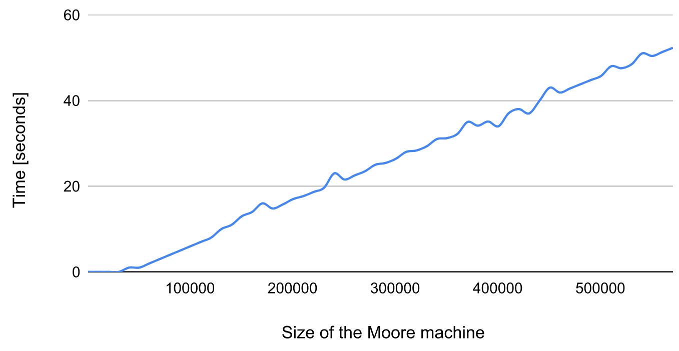

Fig. 14. Monitor synthesis time considering Moore machine as input.

As discussed earlier in the paper, the complexity of our approach is primarily determined by the size of the Moore machine being updated. In previous experiments, we focused on the complete monitor synthesis process, starting from an LTL formula and generating the corresponding partial monitor. However, in Fig. 14, we report the synthesis time when starting directly from a randomly generated Moore machine, as described above. These additional experiments confirm that, as expected, our approach exhibits polynomial complexity relative to the size (*i.e.*, the number of states plus the number of transitions of the Moore machine) of the input Moore machine, in contrast to the exponential growth observed in previous experiments with LTL formulas. These experiments demonstrate that, when considered in a more general setting, our approach consistently produces results in polynomial time.

The other aspect we were interested in analysing was the amount of data consumed by the monitors. Specifically, Fig. 15 reports the results obtained by using standard and partial monitors with respect to the amount of data sent to the monitor for analysis. The x-axis represents the two cases: on the left is when a standard monitor is used (total amount) and on the right is when a partial monitor is used (partial amount). The y-axis represents the amount of data exchanged with the monitor (in KiB). Note that the LTL formulas used for the experiments are randomly generated (via `randltl` Spot's function) and are classified as Safety and Co-Safety properties based on our previous definitions in this paper. This additional separation allows for further analysis of the empirical implications of dropping events for Safety and Co-Safety properties, respectively.

`randltl` is a tool within the Spot library that generates random LTL formulas based on user-defined parameters. It allows users to control the size and complexity of the formulas by specifying the depth, the logical operators to be used, and the probability distribution of these operators. This makes it possible to tailor the generated formulas to specific testing needs. The tool also provides options for reproducibility by allowing users to set a random seed, ensuring the same formulas can be generated across different runs. Thus, `randltl` is particularly useful for benchmarking, testing, and stress-testing tools that work with LTL formulas, as it can generate a wide variety of test cases, including those that might not be covered by manually written examples. It integrates well with other Spot tools, making it a versatile component for exploring and evaluating LTL-related algorithms.

In addition to the properties, the traces given as input to the monitors are also randomly generated. This means that for each random LTL property, a monitor and a partial monitor are synthesised, and then used to analyse a random trace of events.

Now let us focus on the two types of properties analysed in these experiments. Considering Safety properties, as shown in Fig. 15, we can see that using a partial monitor made it possible to drop more than half of the data submitted to the monitor. Similarly, when considering Co-Safety properties, the results are consistent. In fact, we observe that even in this scenario, using partial monitors made it possible to drop more than half of the data to be sent to the monitor. It is worth noting that in our experiments, partial monitors for Safety properties seem to be slightly more efficient at dropping useless data compared to their Co-Safety counterparts.

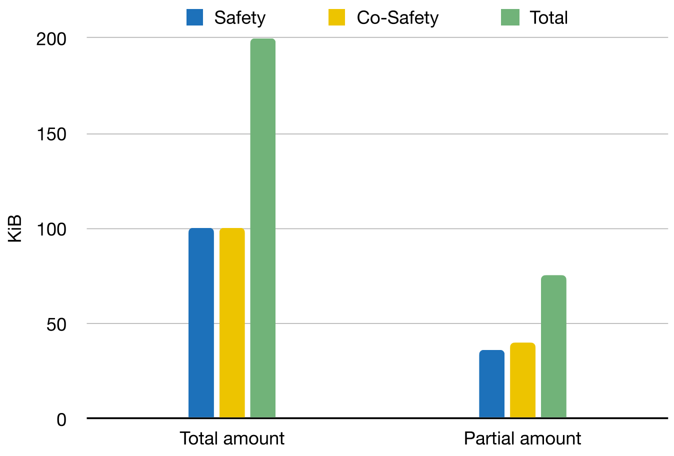

Fig. 15. Amount of data consumed by the standard and the partial monitor.

Before closing the section, we also want to report that the tool has been applied to the remote inspection case study as well. Specifically, we report that the synthesis of the partial monitor (presented in Example 8) took less than 0.05 seconds to conclude. This confirms our observation that even though the technique grows exponentially with the size of the formula given in input, it has no issues at handling formulas such as the on in Example 8.

In our experimental evaluation, we carefully considered both monitorable and non-monitorable properties to assess the performance and scalability of our approach. For the experiments depicted in Fig. 15, we exclusively used monitorable formulas, specifically focusing on safety and co-safety properties, to ensure that the results reflect typical scenarios where monitorability is guaranteed. On the other hand, the experiments illustrated in Fig. 13 included a mix of monitorable and non-monitorable formulas. In these cases, our primary interest was in evaluating the synthesis complexity in relation to the formula size. This allowed us to demonstrate that the synthesis process itself is unaffected by the monitorability of the formula, providing valuable insights into the behaviour of our approach across different types of LTL properties.

7. Related work and discussion

In this work, we presented an intuitive approach to make monitors capable of giving up on properties or events when necessary. As mentioned in Section 3, this is not the first time the notion of monitorability has been studied [8,15,1,17,26]. Nonetheless, regarding related work, we tackle the monitorability problem on a more practical level. Indeed, many works explore the theoretical aspects of what makes a property monitorable, but little has been done to address what we can do with monitorability in practice other than [16], which provides an extensive discussion of how monitorability is useful for practical purposes, and [9], of which this paper represents an extension. For instance, when is it the right time to give up on a property? Which events do we need to keep monitoring? Or, more generally, what can we do with a non-monitorable property? Naturally, there are scenarios where nothing can be done. These are the cases when a property simply cannot be verified at runtime in any possible way, such as the property from Example 7. However, there are scenarios where something can be rightfully concluded, albeit partially. And these are the cases we aim to exploit in this work.

Properties are expected to be fully monitorable (*i.e.*, $\forall_{PZ}$*-monitorable*), because when such a constraint does not hold, we do not have guarantees whether the monitor will ever conclude anything useful. Nonetheless, if the monitor is capable of giving up by recognising and handling ugly prefixes, then non-monitorable properties can be monitored through the use of partial monitors.

Applying such analysis at the monitor level is very important because it not only allows us to give up on the monitor at runtime but also to reuse our approach in various scenarios. Since the approach is based on the Moore machine denoting the monitor (and not the property), it is formalism-agnostic up to a certain level. Thus, we are not just limited to LTL for defining the properties that can be used. We can use another logic as long as a Moore machine can be synthesised. This would not require any modifications to our approach at the theoretical level, but it does require changes in the implementation. For example, either the logic can be converted into LTL if possible, or an automaton representing the monitor needs to be generated; if this automaton is a Moore machine, we are done and ready to use our approach, otherwise, we need to convert it.

From a research perspective, by directly applying our approach on a Moore machine, we also offer a much more reusable workflow. As long as a Moore machine is generated, more challenging aspects can be explored. For instance, Predictive Runtime Verification (PRV) [27–30] can be deployed instead of standard RV. In fact, approaches on PRV of LTL properties exist where a model of the system (a Büchi Automaton) is used to predict future events and to help the monitor to conclude its verdict in advance, before actually observing the events. In such approaches, the flow presented in Fig. 1 is extended to consider the model of the system as well. The important aspect for us to apply that concept to our work is that, even though the workflow is extended, the final result is still a Moore machine, with the additional power of anticipating the conclusive outcomes. Since our approach is directly applied to a Moore machine and not to the property itself, we can still obtain partial monitoring for PRV by analysing the resulting predictive Moore machine. This means we can apply our approach to more challenging scenarios in the future without the need to change anything specific in the process.

In [22], the authors explore the concept of monitorability across both branching and linear time frameworks, focusing on how these different perspectives affect the design and functionality of runtime monitors. They introduce a technique for stopping monitoring as soon as it is recognised that no future events will lead to a verdict. This is achieved by identifying a maximal monitorable fragment of a property and providing a direct, syntax-directed translation that is compositional and avoids the exponential blow-up typically associated with such processes. Their approach is particularly relevant in the branching-time setting, where multi-verdict monitors are inherently unsound, as they demonstrate through formal results. While our work shares the goal of efficient monitoring, it operates within the linear-time domain and focuses on handling non-monitorable properties by extending the monitor to “give up” when it is clear that no conclusive verdict can be reached, rather than immediately terminating upon non-relevant events.

In [31], the same authors further extend the study of partial monitoring by devising a technique to synthesise monitors that are optimal in detecting every monitorable aspect of a logical formula. This work is distinguished by its approach at the level of the formula, where identifying the monitorable subset is fundamental to systematically ensuring that the generated monitors are optimal. In contrast, our approach operates at the level of the automaton, aiming to extend the applicability of runtime verification to properties traditionally considered non-monitorable. While we focus on practical applicability, such as in the case study involving a nuclear inspection robot, we recognise the importance of formal guarantees of monitor optimality, as discussed in [31]. Incorporating such systematic methods to ensure optimality is an important direction for future work as we aim to generalise our approach across different logics and formalisms.

Other works that address monitoring in a partial manner include [32–35]. Unlike our approach, these works are partial in terms of what the monitors can observe of the system. Specifically, they do not assume that all events of the system can be observed by the monitor, but only a subset of them. This results in partial observability of the system by the monitor. Although not explored in this work, the relationship between partial observability and partial monitorability is interesting and their integration could be fruitful.

To summarise, in this paper, we introduce the notion of partial monitoring as a practical view on monitorability, but it is important to note that the theoretical aspects of partial monitorability are different and much harder to tackle [26]. On one hand, we have partial monitoring, where we look into the representation of an existing monitor and identify if it has any states where it should give up. If this is the case and the monitor still has other valid states that are not “give up”, then we have a partial monitor. Partial monitorability, on the other hand, deals with identifying what can make a property partially monitorable. For example, what is the relation of the chosen logic’s operators with monitorability (if any), and how chaining these operators together impacts monitorability, delineating the types of properties that are more amenable and advantageous for partial monitoring, and so on.

8. Conclusions and future work

In this paper, we addressed the challenge of handling monitors generated from non-monitorable properties, proposing methods to extend such monitors to give up when no final verdict can be reached and to focus only on the events necessary for monitoring. We described a practical technique to perform reachability analysis on LTL monitors obtained using standard synthesis approaches [12], demonstrating how the resulting *partial monitors* can avoid getting stuck in scenarios where a final verdict is unattainable, such as when monitoring non-monitorable properties. Our approach was illustrated through a case study in the robotics domain, where a rover inspecting a nuclear facility was partially verified at runtime using non-monitorable properties. We also provided details on the implementation and engineering aspects of a tool to automate the detection and synthesis of partial monitors.

While our work focused on LTL properties, it is important to emphasise that the techniques we propose are not inherently limited to LTL. The choice of LTL allowed us to provide concrete examples that help in illustrating our approach, but the underlying methods are applicable to a broader range of logics.[5]

In future work, we plan to extend our approach to other formalisms such as Metric Temporal Logic (MTL) [37], Signal Temporal Logic (STL) [38], and Runtime Monitoring Language (RML) [39]. This will require updates to our tool, which currently supports only Moore machines, but is necessary to handle more complex scenarios that require more expressive formalisms.

Furthermore, while our approach has shown the effectiveness of synthesising monitors from theoretically non-monitorable properties, particularly in deterministic systems such as the nuclear inspection robot case study, we recognise the challenges in extending this to more dynamic and learning-enabled autonomous systems. These systems, with their machine learning components and adaptable behaviours, present significant challenges due to their non-static nature. Synthesised monitors would need to dynamically adjust to potentially vast and evolving sets of behaviours, complicating or even inhibiting the monitor’s ability to provide meaningful verification. Acknowledging these challenges, future work could explore techniques to enhance monitor adaptability and scalability, potentially integrating predictive runtime verification or leveraging machine learning models to anticipate and respond to changes in system behaviour. This would better accommodate the needs of more complex, learning-enabled systems.

Finally, given that our partial monitors can recognise which events are of interest for verification, we also envisage developing a self-adaptive instrumentation framework. This framework would automatically extract only the events identified as necessary by the partial monitors. While our experiments achieved this event extraction semi-automatically, developing a general-purpose integration mechanism to guide event gathering according to the monitors’ outcomes at runtime would be a valuable advancement.

[5] Specifically, we could update our technique to work on other logics as well, following an approach similar to [36].

CRediT authorship contribution statement

Angelo Ferrando: Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Conceptualization. **Rafael C. Cardoso:** Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: E. Bartocci, Y. Falcone (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics, in: Lecture Notes in Computer Science, vol. 10457, Springer, 2018, pp. 1–33.
[2] E.M. Clarke, Model checking, in: International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 1997, pp. 54–56.
[3] D.W. Loveland, Automated Theorem Proving: a Logical Basis, Fundamental Studies in Computer Science, vol. 6, North-Holland, 1978.
[4] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Methods Program. 78 (5) (2009) 293–303, https://doi.org/10.1016/j.jlap.2008.08.004.
[5] M. Fisher, V. Mascardi, K.Y. Rozier, B. Schlingloff, M. Winikoff, N. Yorke-Smith, Towards a framework for certification of reliable autonomous systems, Auton. Agents Multi-Agent Syst. 35 (1) (2021) 8, https://doi.org/10.1007/s10458-020-09487-2.
[6] M. Fisher, R.C. Cardoso, E.C. Collins, C. Dadswell, L.A. Dennis, C. Dixon, M. Farrell, A. Ferrando, X. Huang, M. Jump, G. Kourtis, A. Lisitsa, M. Luckcuck, S. Luo, V. Pagé, F. Papacchini, M. Webster, An overview of verification and validation challenges for inspection robots, Robotics 10 (2) (2021) 67, https://doi.org/10.3390/ROBOTICS10020067.
[7] J.A. Hertz, A. Krogh, R.G. Palmer, Introduction to the Theory of Neural Computation, the Advanced Book Program, vol. 1, Addison-Wesley, 1991.
[8] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Computational analysis of run-time monitoring - fundamentals of java-mac, Electron. Notes Theor. Comput. Sci. 70 (4) (2002) 80–94, https://doi.org/10.1016/S1571-0661(04)80578-4.
[9] A. Ferrando, R.C. Cardoso, Towards partial monitoring: it is always too soon to give up, in: M. Farrell, M. Luckcuck (Eds.), Proceedings Third Workshop on Formal Methods for Autonomous Systems, FMAS 2021, Virtual, October 21-22, 2021, in: EPTCS, vol. 348, 2021, pp. 38–53.
[10] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society, 1977, pp. 46–57, https://doi.org/10.1109/SFCS.1977.32.
[11] A. Bauer, M. Leucker, C. Schallhart, Monitoring of real-time properties, in: S. Arun-Kumar, N. Garg (Eds.), FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Proceedings, Kolkata, India, December 13-15, 2006, in: Lecture Notes in Computer Science, vol. 4337, Springer, 2006, pp. 260–272, https://doi.org/10.1007/11944836_25.
[12] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for ltl and tltl, ACM Trans. Softw. Eng. Methodol. 20 (4) (Sep. 2011), https://doi.org/10.1145/2000799.2000800.
[13] R. Gerth, D.A. Peled, M.Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: P. Dembinski, M. Sredniawa (Eds.), Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995, in: IFIP Conference Proceedings, vol. 38, Chapman & Hall, 1995, pp. 3–18, https://doi.org/10.1007/978-0-387-34892-6_1.
[14] M.O. Rabin, D.S. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (2) (1959) 114–125, https://doi.org/10.1147/rd.32.0114.
[15] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, An operational guide to monitorability, in: P.C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods - 17th International Conference, Proceedings, SEFM 2019, Oslo, Norway, September 18-20, 2019, in: Lecture Notes in Computer Science, vol. 11724, Springer, 2019, pp. 433–453, https://doi.org/10.1007/978-3-030-30446-1_23.
[16] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, An operational guide to monitorability with applications to regular properties, Softw. Syst. Model. 20 (2) (2021) 335–361, https://doi.org/10.1007/S10270-020-00860-Z.
[17] A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Proceedings, Hamilton, Canada, August 21-27, 2006, in: Lecture Notes in Computer Science, vol. 4085, Springer, 2006, pp. 573–586, https://doi.org/10.1007/11813040_38.
[18] Z. Chen, Y. Wu, O. Wei, B. Sheng, Deciding weak monitorability for runtime verification, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 163–164, https://doi.org/10.1145/3183440.3195077.
[19] T.A. Henzinger, N.E. Saraç, Monitorability under assumptions, in: J. Deshmukh, D. Nickovic (Eds.), Runtime Verification - 20th International Conference, Proceedings, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, in: Lecture Notes in Computer Science, vol. 12399, Springer, 2020, pp. 3–18, https://doi.org/10.1007/978-3-030-60508-7_1.
[20] B. Alpern, F.B. Schneider, Recognizing safety and liveness, Distrib. Comput. 2 (3) (1987) 117–126, https://doi.org/10.1007/BF01782772.
[21] A.P. Sistla, Safety, liveness and fairness in temporal logic, Form. Asp. Comput. 6 (5) (1994) 495–512, https://doi.org/10.1007/BF01211865.
[22] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, Adventures in monitorability: from branching to linear time and back again, Proc. ACM Program. Lang. 3 (POPL) (2019) 52:1–52:29, https://doi.org/10.1145/3290365.
[23] A. Francalanza, L. Aceto, A. Ingólfsdóttir, Monitorability for the Hennessey-Milner logic with recursion, Form. Methods Syst. Des. 51 (1) (2017) 87–116, https://doi.org/10.1007/s10703-017-0273-z.
[24] K.G. Larsen, Proof systems for satisfiability in Hennessey-Milner logic with recursion, Theor. Comput. Sci. 72 (2&3) (1990) 265–288, https://doi.org/10.1016/0304-3975(90)90038-J.
[25] T. Wright, A. West, M. Licata, N. Hawes, B. Lennox, Simulating ionising radiation in gazebo for robotic nuclear inspection challenges, Robotics 10 (3) (2021), https://doi.org/10.3390/robotics10030086.
[26] L. Ciccone, F. Dagnino, A. Ferrando, Ain't no stopping us monitoring now, CoRR, arXiv:2211.11544 [abs], 2022, https://doi.org/10.48550/ARXIV.2211.11544.
[27] X. Zhang, M. Leucker, W. Dong, Runtime verification with predictive semantics, in: NASA Formal Methods, in: LNCS, vol. 7226, Springer, 2012, pp. 418–432, https://doi.org/10.1007/978-3-642-28891-3_37.
[28] M. Leucker, Sliding between model checking and runtime verification, in: Runtime Verification, in: LNCS, vol. 7687, Springer, 2012, pp. 82–87, https://doi.org/10.1007/978-3-642-35632-2_10.
[29] A. Ferrando, R.C. Cardoso, M. Farrell, M. Luckcuck, F. Papacchini, M. Fisher, V. Mascardi, Bridging the gap between single- and multi-model predictive runtime verification, Form. Methods Syst. Des. 59 (1) (2021) 44–76, https://doi.org/10.1007/S10703-022-00395-7.
[30] A. Ferrando, G. Delzanno, Incrementally predictive runtime verification, J. Log. Comput. 33 (4) (2023) 796–817, https://doi.org/10.1093/LOGCOM/EXAD012.

[31] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, K. Lehtinen, The best a monitor can do, in: C. Baier, J. Goubault-Larrecq (Eds.), 29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia, Virtual Conference, in: LIPIcs, vol. 183, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 7:1–7:23, https://doi.org/10.4230/LIPICS.CSL.2021.7.

[32] D. Ancona, A. Ferrando, V. Mascardi, Mind the gap! Runtime verification of partially observable mass with probabilistic trace expressions, in: D. Baumeister, J. Rothe (Eds.), Multi-Agent Systems - 19th European Conference, Proceedings, EUMAS 2022, Düsseldorf, Germany, September 14-16, 2022, in: Lecture Notes in Computer Science, vol. 13442, Springer, 2022, pp. 22–40, https://doi.org/10.1007/978-3-031-20614-6_2.

[33] A. Cimatti, C. Tian, S. Tonetta, Assumption-based runtime verification, Form. Methods Syst. Des. 60 (2) (2022) 277–324, https://doi.org/10.1007/S10703-023-00416-Z.

[34] R. Taleb, R. Khoury, S. Hallé, Runtime verification under access restrictions, in: S. Bliudze, S. Gnesi, N. Plat, L. Semini (Eds.), 9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021, IEEE, 2021, pp. 31–41, https://doi.org/10.1109/FORMALISE52586.2021.00010.

[35] H. Kallwies, M. Leucker, C. Sánchez, Symbolic runtime verification for monitoring under uncertainties and assumptions, in: A. Bouajjani, L. Holík, Z. Wu (Eds.), Automated Technology for Verification and Analysis - 20th International Symposium, Proceedings, ATVA 2022, Virtual Event, October 25-28, 2022, in: Lecture Notes in Computer Science, vol. 13505, Springer, 2022, pp. 117–134, https://doi.org/10.1007/978-3-031-19992-9_8.

[36] Y. Falcone, J. Fernandez, L. Mounier, What can you verify and enforce at runtime?, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 349–382, https://doi.org/10.1007/s10009-011-0196-8.

[37] R. Koymans, Specifying real-time properties with metric temporal logic, Real-Time Syst. 2 (4) (1990) 255–299, https://doi.org/10.1007/BF01995674.

[38] O. Maler, D. Nickovic, Monitoring temporal properties of continuous signals, in: Y. Lakhnech, S. Yovine (Eds.), Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, Proceedings, FTRTFT 2004, Grenoble, France, September 22-24, 2004, in: Lecture Notes in Computer Science, vol. 3253, Springer, 2004, pp. 152–166, https://doi.org/10.1007/978-3-540-30206-3_12.

[39] D. Ancona, L. Franceschini, A. Ferrando, V. Mascardi, RML: theory and practice of a domain specific language for runtime verification, Sci. Comput. Program. 205 (2021) 102610, https://doi.org/10.1016/j.scico.2021.102610.