Towards Partial Monitoring: It is Always too Soon to Give Up*

Angelo Ferrando
University of Genova
Genova, Italy
angelo.ferrando@unige.it

Rafael C. Cardoso
The University of Manchester
Manchester, United Kingdom
rafael.cardoso@manchester.ac.uk

Runtime Verification is a lightweight formal verification technique. It is used to verify at runtime whether the system under analysis behaves as expected. The expected behaviour is usually formally specified by means of properties, which are used to automatically synthesise monitors. A monitor is a device that, given a sequence of events representing a system execution, returns a verdict symbolising the satisfaction or violation of the formal property. Properties that can (resp. cannot) be verified at runtime by a monitor are called *monitorable* and *non-monitorable*, respectively. In this paper, we revise the notion of monitorability from a practical perspective, where we show how *non-monitorable* properties can still be used to generate *partial* monitors, which can partially check the properties. Finally, we present the implications both from a theoretical and practical perspectives.

1 Introduction

Runtime Verification (RV) is a well-known lightweight formal verification technique [5]. Similar to other existing formal verification techniques, such as Model Checking [9] and Theorem Proving [21], it aims to verify the system behaviour, usually referred to as the System Under Analysis (SUA). Such a system can be composed of both software and hardware components, and the formal verification technique of choice is used to verify that everything works as expected.

RV achieves the verification of the SUA through monitoring. Specifically, starting from a formal property expressed in some formalism of choice, one (or multiple) monitors are generated. A monitor is a device that, given a sequence of events (a trace) that are generated by the system execution, it verifies the conformance of such a trace with respect to the formal property. Since the trace can be generated at runtime, the monitor can inform the system's users about unexpected behaviours; *i.e.*, events which violate the formal specification.

Differently from other formal verification techniques, RV is performed on the execution of the system. The formal properties are verified on traces of events generated by actual system executions. This is an important difference with respect to more traditional formal verification techniques such as Model Checking, where the verification is performed statically over an abstracted model of the system. RV does not require any model, nor any other information apart from execution traces, which makes it well suited to be used in *black-box* scenarios where not much is known about the SUA, such as in autonomous systems. Moreover, from a computational perspective RV performs better than traditional verification techniques; since monitors only take as input what the SUA produces, without the need of going through a model. It has been shown that RV offers polynomial time behaviour with respect to the length of the analysed trace [20].

Providing certification for reliable autonomous systems is not an easy task [11]. Formal verification of monolithic systems is already hard; to apply such techniques in the context of autonomous, cyber-physical, or even robotic systems makes it even more complicated. This is mainly due to the fact that

*Cardoso's work supported by Royal Academy of Engineering under the Chairs in Emerging Technologies scheme.

M. Farrell & M. Luckcuck (Eds.):
Third Workshop on Formal Methods
for Autonomous Systems (FMAS2021)
EPTCS 348, 2021, pp. 38–53, doi:10.4204/EPTCS.348.3

these systems are intrinsically unpredictable [22]; especially when Machine Learning techniques are involved, such as Neural Networks [15]. In these scenarios, RV may be of help, especially since it does not require a model of the system and it can be deployed at runtime while the system is still running. In this way, even though the system offers some unpredictable aspects, by adding monitors it is possible to improve the system reliability. In fact, since monitors can be deployed together with the system, the monitors will be there to detect and possibly react accordingly in case of unexpected behaviours (*e.g.*, by triggering or implementing some mechanism to handle such failures).

Unfortunately, some formal properties cannot be monitored at runtime. A formal property is considered *monitorable*, when it is possible to synthesise a monitor which verifies it. In turn, a formal property is denoted as *non-monitorable*, when it is not possible to synthesise such a monitor. In literature, there exists different definitions on the requirements for a property to be considered monitorable [16]. Nonetheless, the most common requirement for a property to be monitorable is that it should always be possible to conclude the satisfaction or violation of the property. Which means that there should not exist traces of events which make the property never satisfied nor violated. The reason such kind of properties are usually avoided is that the resulting monitors might be useless, since they may never be able to conclude anything about the SUA. Because of this, RV approaches usually focus on monitorable fragments of the properties to analyse. However, as we will show in this paper, some of these non-monitorable properties may still be worth to be analysed at runtime, even though if only partially. Therefore, we pose the following research question:

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

We suggest that a viable answer to this question is using *partial monitoring*. Since a non-monitorable property does not give any assurance on satisfaction or violation, we need to synthesise monitors which are capable of *giving up* on the verification process when it is clear that it will never arrive at any conclusion. This is especially relevant in the context of autonomous systems, where the availability of computational power and memory might be limited (*e.g.*, in embedded systems). In such context, the presence of a component (the monitor) that arrives at a certain state where it does not do anything useful anymore is not only pointless, but it is also a waste of resources which could be better allocated. Thus, it is important to have monitors capable of giving up on the verification of a property, in order to safely reclaim otherwise used resources. At the same time, by using partial monitors we can be less restrictive on the kind of formal properties we are allowed to use, since a non-monitorable property can still be partially monitored at runtime.

As a proof of concept, in this work we exemplify our approach in a robotic application where an autonomous rover is deployed into a nuclear facility to perform remote inspection. In this scenario, the dynamic environment makes it hard to formally verify properties using traditional formal methods such as model checking. Thus, we can use RV to formally verify at runtime how the rover behaves. Using this application, we show that non-monitorable formal properties have parts of it that can still be verified using a partial monitor, as long as the monitor is capable of detecting when to give up. For example, we can have partial monitors to detect that the rover does not stay in areas with high-level of radiation, but if it observes a bad event (*e.g.*, an event that would cause the monitor to be stuck in inconclusive states), which would render the monitor useless, then it needs to give up.

In this paper, we briefly revise the notion of *monitorability* [16, 2, 5, 25], where we focus more on its engineering implications for what concerns the monitor synthesis. To do so, we present a straightforward extension of the standard monitor synthesis for temporal properties in Linear Temporal Logic (LTL), where we take into consideration that a monitor could fail to completely verify an LTL property. We show how we can achieve this reasoning at the monitor level, instead of the more standard way of doing

this at the property specification level. Specifically, we reduce the problem of a monitor recognising when to give up to a reachability problem inside the monitor representation.

The remainder of this paper is structured as follows. Section 2 presents some background definitions and notation that are used in the paper. Section 3 revises the notion of monitorability and reports related works in literature. Section 4 proposes our contribution, where the notion of partial monitoring is introduced. Section 4.1 demonstrates the use of partial monitors in an autonomous rover performing remote inspection tasks. In Section 4.2, we give the details on how the approach described in this paper has been implemented. Section 5 discusses the approach and its engineering implications in the monitor synthesis. Finally, Section 6 concludes the paper with final remarks and future research directions.

2 Preliminaries

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.

The standard formalism to specify formal properties in RV is propositional Linear Temporal Logic (LTL [24]). The relevant parts of the syntax of LTL are as follows:

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

where $ev \in \Sigma$ is an event (a proposition), φ is a formula, $\mathbf{U}$ stands for *until*, and $\bigcirc$ 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:

$$\begin{array}{lcl}
\sigma & \models & ev \text{ if } ev \in \sigma(0) \\
\sigma & \models & \neg\varphi \text{ if } \sigma \not\models \varphi \\
\sigma & \models & \varphi \wedge \varphi' \text{ if } \sigma \models \varphi \text{ and } \sigma \models \varphi' \\
\sigma & \models & \varphi \vee \varphi' \text{ if } \sigma \models \varphi \text{ or } \sigma \models \varphi' \\
\sigma & \models & \bigcirc\varphi \text{ if } \sigma^1 \models \varphi \\
\sigma & \models & \varphi \, \mathbf{U} \, \varphi' \text{ if } \exists_{i \geq 0}. \sigma^i \models \varphi' \text{ and } \forall_{0 \leq j < i}. \sigma^j \models \varphi
\end{array}$$

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 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 $Mon_\varphi : \Sigma^* \to \mathbb{B}_3$, where $\mathbb{B}_3 = \{\top, \bot, ?\}$:*

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

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, 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 is applied while the system is still running, and not all information about it are 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 (FSM where the output value of a state is only determined by the state) [6, 7]. 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, $\delta : Q \times \Sigma \to Q$ is the transition function mapping a state and an event to the next state, and $\gamma : Q \to O$ is the function mapping a state to the output alphabet.

In [7], Bauer *et al.* present the sequence of steps required to generate from an LTL formula φ the corresponding Moore machine instantiating the Mon_φ function (as summarised in Figure 1).

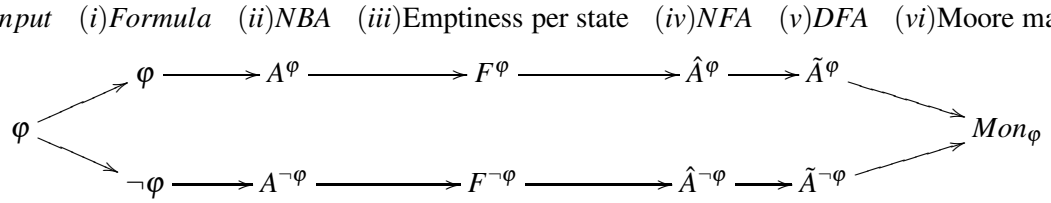


Figure 1: Steps required to generate an FSM from an LTL formula φ. NBA is Non-deterministic Büchi Automaton, NFA is Non-deterministic Finite Automaton, and DFA is Deterministic Finite Automaton.

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 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 [26]) 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 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_φ is reported in Figure 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 positively φ since there is no continuation capable of violating φ (i.e., the corresponding state in $\tilde{A}^{\neg\varphi}$ is not final).*

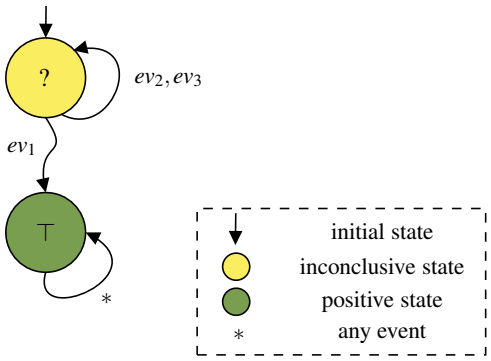


Figure 2: Moore machine instantiated for the monitor generated by φ of Example 1.

3 Monitorability

Monitorability [16] 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 at runtime of formal properties. However, the level of detail such notion is defined in the literature may vary. It has a wide range of definitions, some are more restrictive, while others are more flexible. A thorough presentation of the existing variations of monitorability can be found in [2]; 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 [25], where the concept of monitorability was generalised w.r.t. its first appearance [16]. We chose their view of monitoriability 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 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* [8].

Definition 3 *A property φ is (existentially Pnueli-Zaks) $\exists_{PZ}$-monitorable if it is 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; e.g., 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. Figure 3 reports the monitor obtained by φ.*

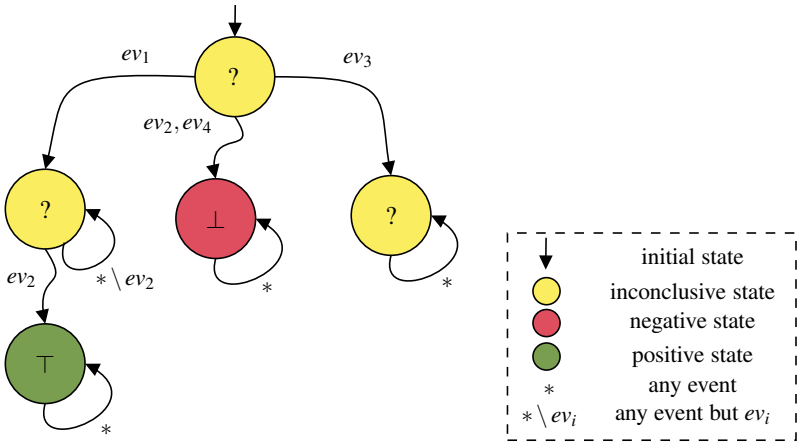


Figure 3: Moore machine of the $\exists_{PZ}$-*monitorable* property φ presented in Example 2.

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); while on the right branch, we have the possibility to violate the property by observing something different from ev_4 (negative). Figure 4 reports the monitor obtained by φ.

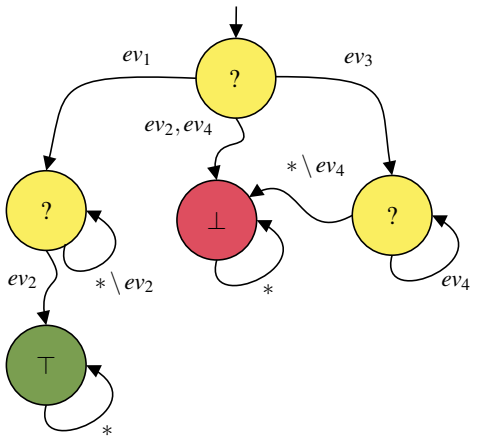


Figure 4: Monitor (as Moore machine) of the $\forall_{PZ}$-*monitorable* property φ presented in Example 3.

Monitorable properties are defined according to Definition 4 in a variety of past research [7, 5, 20, 14, 16]. 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 [7] 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 [3]. 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 [27].

Figure 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.

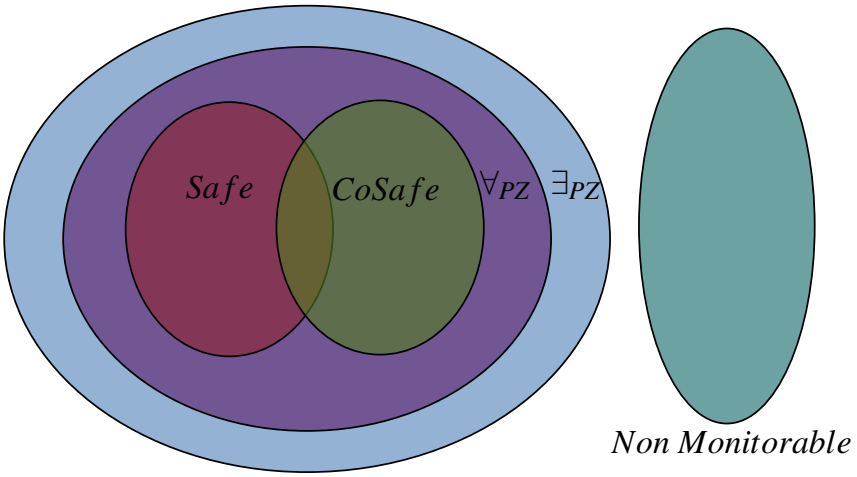


Figure 5: Hierarchy of classes of monitorable properties.

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 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 [25], 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 [1, 12], the notion of monitorability is presented for linear and branching flavours of Hennessy-Milner Logic with recursion (recHML) [18]. 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 [14], 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 the notion of partial monitoring from a practical perspective. First, we formally define partial monitors and show an example of how it works based on a property from a previous example. Then, we discuss the application and benefits of partial monitoring to an example of an autonomous rover performing remote inspection tasks. Finally, we present the implementation details of our tool, which is capable of automatically synthesising partial monitors from existing monitors for non-monitorable properties, and discuss the engineering aspects of our approach.

In Definition 1, we had 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".

Definition 7 (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* $Mon_\varphi : \Sigma^* \to \mathbb{B}_4$*, where* $\mathbb{B}_4 = \{\top, \bot, ?, \chi\}$*:*

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

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

Definition 7 presents the notion of a *partial 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.

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. Figure 6 reports the partial monitor obtained by updating the monitor from Figure 3.*

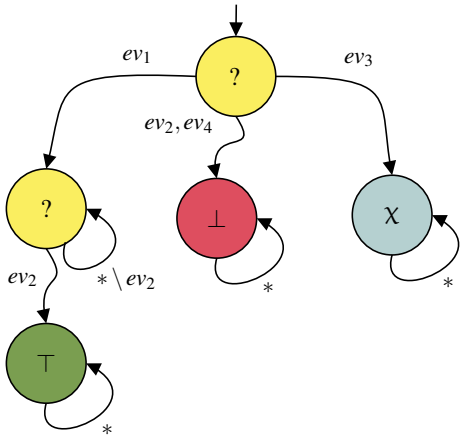


Figure 6: Partial monitor of the $\exists_{PZ}$-*monitorable* property φ presented in Example 2. Here, we can note how the previously inconclusive state on the right has now become a "give up" state (grey colour).

Given a standard monitor, to obtain a partial monitor we add an additional post-processing step after generating the Moore machine. 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. Specifically, in the worst case scenario the time complexity is $O(N^2 + NE)$, with N the number of states, and E the number of edges. But, to have a better understanding of the time complexity, it is important to note the relation between N and φ. In fact, the standard LTL monitor synthesis procedure (Figure 1) generates a Moore machine with double exponential size w.r.t. the size of the LTL property $|\varphi|$. Thus, if we expand the time complexity w.r.t. to the property, we obtain that the approach has time complexity $O(2^{2^{|\varphi|+1}} + 2^{2^{|\varphi|}}E)$. Nonetheless, properties are usually small in size, which makes the time complexity less limiting.

4.1 A Remote Inspection Example

We demonstrate the usefulness of our approach by applying it to the remote inspection example mentioned in the introduction. This example is based on a simulation, first introduced in [28], 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 an integrity analysis (*e.g.*, deterioration of the container). The objective of the rover is to patrol and inspect important locations (*i.e.*, waypoints) around the facility. As part of the inspection of a particular location, the rover has to take measurements of the radiation level when it arrives in such location.

Example 5 *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 Figure 7, with Figure 7a containing the traditional Moore machine, and then after applying our technique we can see in Figure 7b 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).*

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

(b) New monitor generated after applying our approach.

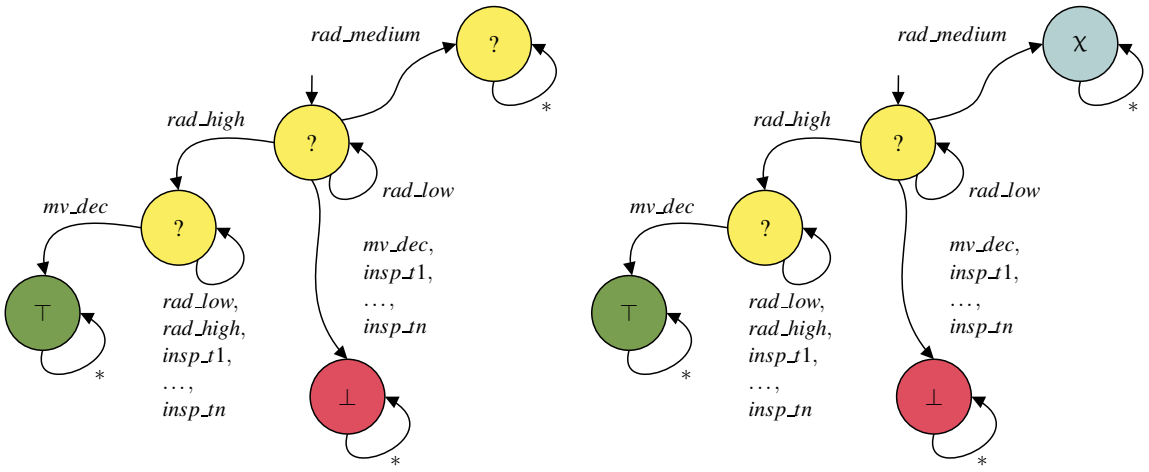
Figure 7: A simple property $\varphi = \Box\Diamond inspect\_tank\_1$.

Example 6 *Next, let us consider a more interesting property where we can demonstrate that partial monitoring can indeed be useful:*

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

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 (i.e., high level of radiation can be dangerous to the rover). Otherwise, if we observe radiation_medium, then we have to patrol one of the radiation tanks $(1 \ldots n)$ *to identify if there are any abnormalities.*

The monitor (represented as a Moore machine) for this more complex property is shown in Figure 8. This monitor originally has three inconclusive states, as shown in Figure 8a. 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,



(a) Monitor (as Moore machine) for the property. (b) Updated monitor after applying our technique.

Figure 8: 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.

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. Figure 8b contains the result of this reachability analysis.

4.2 Implementation

We made a Java program[1] which implements the transformation from standard to partial monitor. This tool depends on LamaConv[2], a Java library that is capable of translating expressions in temporal logics into equivalent automata, and to generate monitors out of these automata. These monitors can then 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.

[1] `https://github.com/AngeloFerrando/PartialMonitor`
[2] `https://www.isp.uni-luebeck.de/lamaconv`

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 χ (give up) states.

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 the only change 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). 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.

5 Discussion

In this work, we presented an intuitive approach to make monitors capable of giving up when necessary. As mentioned in Section 3, this is not the first time the notion of monitorability has been studied. Nonetheless, w.r.t. related work, we tackle the monitorability problem on a more practical level. Indeed, there are plenty of works which study and explore the theoretical aspects of what makes a property monitorable; but, not much has been done to answer what we can do with monitorability in practice. For instance, when is it the right time to give up on a property? 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 5). However, there are scenarios where something can be rightfully concluded, albeit partially. And these are the cases we aimed to exploit in this work. In general, properties are expected to be fully monitorable (*i.e.*, $\forall_{PZ}$-$monitorable$), because when such 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 this does not only allow 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) [29, 19] can be deployed instead of standard RV. In fact, works 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 works, the flow presented in Figure 1 is extended for considering the model of the system as well. The important aspect for us 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 on a Moore machine and not on 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.

To summarise, in this paper we introduced 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. On one hand we have partial monitoring, where we look into the representation of an existing monitor and we 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, would deal with identifying what can make a property partially monitorable; *e.g.,* what is the relation of the chosen logic's operators with monitorability (if any) and how chaining these operators together impacts monitorability, delineate the types of properties that are more amenable and advantageous for partial monitoring, and so on.

6 Conclusions and Future Work

In this paper, we discussed the issue of handling monitors generated from non-monitorable properties and how such monitors can be extended to give up when no final verdict can be ever concluded. We described a practical technique to perform reachability analysis of LTL monitors obtained using standard synthesis approaches [7], and how the resulting *partial monitors* can avoid to get stuck in scenarios where no final verdict can be concluded, such as when trying to monitor non-monitorable properties. We have demonstrated the use of partial monitoring in an example from the robotics domain, where a rover performing the inspection of a nuclear facility can be partially verified at runtime using non-monitorable properties. We also described the implementation details and engineering aspects of a tool that we developed to automate the detection and synthesis of partial monitors.

This work has focused on the engineering and implications of monitorability for the synthesis of partial monitors. In future work, we plan to extend the approach to other formalisms, such as Metric Temporal Logic (MTL) [17], Signal Temporal Logic (STL) [23], Runtime Monitoring Language (RML) [4], and so on. We also want to integrate our approach with existing RV tools for autonomous systems such as ROSMonitoring [10], which is a RV framework for the popular middleware for robotic development Robot Operating System (ROS)[3]. Adding support to other formalisms requires updates to the tool, since right now it can only take as input plain Moore machines. This is important since more complex scenarios may require more expressive formalisms than LTL, and the issue of having partial monitors (as well as monitorability) arises in other formalisms as well.

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2019): *Adventures in monitorability: from branching to linear time and back again*. Proc. ACM Program. Lang. 3(POPL), pp. 52:1–52:29, doi:10.1145/3290365.

[2] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2019): *An Operational Guide to Monitorability*. In Peter Csaba Ölveczky & Gwen Salaün, editors: *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway,*

[3] `https://www.ros.org/`

September 18-20, 2019, Proceedings, *Lecture Notes in Computer Science* 11724, Springer, pp. 433–453, doi:10.1007/978-3-030-30446-1_23.

[3] Bowen Alpern & Fred B. Schneider (1987): *Recognizing Safety and Liveness*. *Distributed Comput.* 2(3), pp. 117–126, doi:10.1007/BF01782772.

[4] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2021): *RML: Theory and practice of a domain specific language for runtime verification*. *Sci. Comput. Program.* 205, p. 102610, doi:10.1016/j.scico.2021.102610.

[5] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In Ezio Bartocci & Yliès Falcone, editors: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[6] Andreas Bauer, Martin Leucker & Christian Schallhart (2006): *Monitoring of Real-Time Properties*. In S. Arun-Kumar & Naveen Garg, editors: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, *Lecture Notes in Computer Science* 4337, Springer, pp. 260–272, doi:10.1007/11944836_25.

[7] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL*. *ACM Trans. Softw. Eng. Methodol.* 20(4), doi:10.1145/2000799.2000800.

[8] Zhe Chen, Yifan Wu, Ou Wei & Bin Sheng (2018): *Deciding weak monitorability for runtime verification*. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik & Mark Harman, editors: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, ACM, pp. 163–164, doi:10.1145/3183440.3195077.

[9] Edmund M Clarke (1997): *Model checking*. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, pp. 54–56, doi:10.1016/0304-3975(85)90223-3.

[10] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini & Viviana Mascardi (2020): *ROSMonitoring: A Runtime Verification Framework for ROS*. In: *Towards Autonomous Robotic Systems*, Springer International Publishing, Cham, pp. 387–399, doi:10.1007/978-3-030-63486-5_40.

[11] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff & Neil Yorke-Smith (2021): *Towards a framework for certification of reliable autonomous systems*. *Auton. Agents Multi Agent Syst.* 35(1), p. 8, doi:10.1007/s10458-020-09487-2.

[12] Adrian Francalanza, Luca Aceto & Anna Ingólfsdóttir (2017): *Monitorability for the Hennessy-Milner logic with recursion*. *Formal Methods Syst. Des.* 51(1), pp. 87–116, doi:10.1007/s10703-017-0273-z.

[13] Rob Gerth, Doron A. Peled, Moshe Y. Vardi & Pierre Wolper (1995): *Simple on-the-fly automatic verification of linear temporal logic*. In Piotr Dembinski & Marek Sredniawa, editors: *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*, *IFIP Conference Proceedings* 38, Chapman & Hall, pp. 3–18, doi:10.1007/978-0-387-34892-6_1.

[14] Thomas A. Henzinger & N. Ege Saraç (2020): *Monitorability Under Assumptions*. In Jyotirmoy Deshmukh & Dejan Nickovic, editors: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, *Lecture Notes in Computer Science* 12399, Springer, pp. 3–18, doi:10.1007/978-3-030-60508-7_1.

[15] John A. Hertz, Anders Krogh & Richard G. Palmer (1991): *Introduction to the theory of neural computation*. *The advanced book program* 1, Addison-Wesley.

[16] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky & Mahesh Viswanathan (2002): *Computational Analysis of Run-time Monitoring - Fundamentals of Java-MaC*. *Electron. Notes Theor. Comput. Sci.* 70(4), pp. 80–94, doi:10.1016/S1571-0661(04)80578-4.

[17] Ron Koymans (1990): *Specifying Real-Time Properties with Metric Temporal Logic*. *Real Time Syst.* 2(4), pp. 255–299, doi:10.1007/BF01995674.

[18] Kim Guldstrand Larsen (1990): *Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion*. *Theor. Comput. Sci.* 72(2&3), pp. 265–288, doi:10.1016/0304-3975(90)90038-J.

[19] Martin Leucker (2012): *Sliding between Model Checking and Runtime Verification*. In: *Runtime Verification*, *LNCS* 7687, Springer, pp. 82–87, doi:10.1007/978-3-642-35632-2_10.

[20] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *J. Log. Algebraic Methods Program.* 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[21] Donald W. Loveland (1978): *Automated theorem proving: a logical basis*. *Fundamental studies in computer science* 6, North-Holland.

[22] Joseph B. Lyons, Matthew A. Clark, Alan R. Wagner & Matthew J. Schuelke (2017): *Certifiable Trust in Autonomous Systems: Making the Intractable Tangible*. *AI Mag.* 38(3), pp. 37–49, doi:10.1609/aimag.v38i3.2717.

[23] Oded Maler & Dejan Nickovic (2004): *Monitoring Temporal Properties of Continuous Signals*. In Yassine Lakhnech & Sergio Yovine, editors: *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, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, *Lecture Notes in Computer Science* 3253, Springer, pp. 152–166, doi:10.1007/978-3-540-30206-3_12.

[24] Amir Pnueli (1977): *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, pp. 46–57, doi:10.1109/SFCS.1977.32.

[25] Amir Pnueli & Aleksandr Zaks (2006): *PSL Model Checking and Run-Time Verification Via Testers*. In Jayadev Misra, Tobias Nipkow & Emil Sekerinski, editors: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, *Lecture Notes in Computer Science* 4085, Springer, pp. 573–586, doi:10.1007/11813040_38.

[26] Michael O. Rabin & Dana S. Scott (1959): *Finite Automata and Their Decision Problems*. *IBM J. Res. Dev.* 3(2), pp. 114–125, doi:10.1147/rd.32.0114.

[27] A. Prasad Sistla (1994): *Safety, Liveness and Fairness in Temporal Logic*. *Formal Aspects Comput.* 6(5), pp. 495–512, doi:10.1007/BF01211865.

[28] Thomas Wright, Andrew West, Mauro Licata, Nick Hawes & Barry Lennox (2021): *Simulating Ionising Radiation in Gazebo for Robotic Nuclear Inspection Challenges*. *Robotics* 10(3), doi:10.3390/robotics10030086.

[29] Xian Zhang, Martin Leucker & Wei Dong (2012): *Runtime Verification with Predictive Semantics*. In: *NASA Formal Methods*, *LNCS* 7226, Springer, pp. 418–432, doi:10.1007/978-3-642-28891-3_37.