Runtime Verification via Rational Monitor with Imperfect Information

ANGELO FERRANDO, University of Modena and Reggio Emilia, Italy
VADIM MALVONE, Telecom Paris, Institut Polytechnique de Paris, France

Trusting software systems, particularly autonomous ones, is challenging. To address this, formal verification techniques can ensure these systems behave as expected. Runtime Verification (RV) is a leading, lightweight method for verifying system behaviour during execution. However, traditional RV assumes perfect information, meaning the monitoring component perceives everything accurately. This assumption often fails, especially with autonomous systems operating in real-world environments where sensors might be faulty. Additionally, traditional RV considers the monitor to be passive, lacking the capability to interpret the system's information and thus unable to address incomplete data. In this work, we extend standard RV of Linear Temporal Logic properties to accommodate scenarios where the monitor has imperfect information and behaves rationally. We outline the necessary engineering steps to update the verification pipeline and demonstrate our implementation in a case study involving robotic systems.

CCS Concepts: • **Theory of computation** → *Logic and verification*; *Formal languages and automata theory*; • **Software and its engineering** → *Formal software verification.*

Additional Key Words and Phrases: Runtime Verification, Autonomous Systems, Imperfect Information, Rational Monitor

ACM Reference Format:
Angelo Ferrando and Vadim Malvone. 2018. Runtime Verification via Rational Monitor with Imperfect Information. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Developing high-quality software is becoming increasingly challenging due to the rise of distributed and autonomous systems, which exhibit inherent complexity and dynamic, unpredictable behaviours [29]. Unlike traditional monolithic systems, these modern architectures require new approaches to engineering and verification, which have to ensure reliability and safety, that cope with such software evolution. In this context, the software evolution refers to the shift towards decentralised, interconnected, and autonomous designs, which introduce challenges such as asynchronous communication, decentralised control, and autonomous decision making. To address these, verification techniques like Runtime Verification (RV) must adapt. Specifically, RV must evolve to monitor distributed components in real time, handle the unpredictability of autonomous behaviours, and scale effectively in complex environments. This paper explores how RV can meet these challenges, bridging the gap between software evolution and verification needs.

Runtime Verification [3, 28] is a formal verification technique that is used to verify the runtime behaviour of software and hardware systems. Unlike other verification methods, RV is not exhaustive; it focuses solely on the system's actual execution. This means that a violation of expected behaviour is only detected if it occurs in the execution trace. Despite

Authors' Contact Information: Angelo Ferrando, angelo.ferrando@unimore.it, University of Modena and Reggio Emilia, Modena, Italy, Italy; Vadim Malvone, vadim.malvone@telecom-paris.fr, Telecom Paris, Institut Polytechnique de Paris, Palaiseau, France.

Manuscript submitted to ACM

this, RV is a lightweight technique because it does not check all possible system behaviours, allowing it to scale better than static verification methods, which often struggle with the state space explosion problem.

RV emerged after static verification methods like model checking [13], inheriting much from them, particularly in specifying the formal properties to verify. One of the most widely used formalisms in model checking, and consequently in RV, is Linear Temporal Logic (LTL) [30]. While we will detail its syntax and semantics later in the paper, we focus on LTL's implicit assumption of perfect information about the system. Typically, LTL verification assumes that the system under analysis provides all necessary information for verification [10]. This is reflected at the verification level by generating atomic propositions that denote the knowledge about the system, which are used to verify the properties of interest. However, this assumption does not always hold, particularly for systems with autonomous, distributed, or faulty components, such as faulty sensors in real-world environments. In such cases, assuming all necessary information is available is overly optimistic. Although other works have addressed LTL RV with imperfect information [4, 5, 25, 27, 32], this research line is the first to tackle the problem fundamentally without requiring a new verification pipeline. Specifically, in [18], we presented an initial attempt to extend the standard monitor synthesis pipeline to explicitly account for imperfect information. Building on this idea, we further refine our approach to incorporate the monitor's ability to be rational, enhancing its capability to handle imperfect information. While our previous work focused on engineering a monitor to provide correct answers despite imperfect information, our current work refines the monitor's information processing to yield conclusive results.

The contribution of this paper is twofold. First, we formally define the notion of imperfect information with respect to the monitor's visibility over the system, and re-engineer the LTL monitor's synthesis pipeline to recognise this visibility information. Second, we introduce the concept of a rational monitor, which can dynamically manage its visibility. In particular, we define two classes of rational monitors: *active* and *reactive*. The first one corresponds to an imperfect information monitor, which is able to reason upon the resources that it needs to verify its own formal specification. The second one, instead, refines the same idea but allowing the monitor to reactively update its knowledge at execution time. To the best of our knowledge, this notion has not been previously defined in the literature. Additionally, we present the details of the prototype implemented to support our claims, providing the community with an LTL monitoring library that natively supports imperfect information and rationality. Finally, we show the use of this prototype in a case study.

Differences with the conference version. This contribution differs from its conference version [18] in what follows. First of all, the case study (Section 3) has been completely revisited and extended (making it running example). Specifically, we revised existing formulas and defined new ones. In addition to the case study, all the examples in the paper have been properly revisited, while additional ones have been added. Furthermore, we have extended the imperfect information section by adding novel definitions and enhancing the all-presentation (Section 4). Then, the introduction of rational monitors and the resulting approach is completely novel to this work (Section 5). Finally, the implementation (Section 6) has been extended to take into consideration the new rational aspects (along with its additional parameters). On top of the implementation, the experiments have been completely revisited w.r.t. the new case study (Section 6.1.2) and the metrics (Section 6.1.3) considered in this work. Lastly, the related work has been completely revisited (Section 7).

Expected real-world implications of uncertainty in runtime verification. Runtime verification with uncertainty has promising real-world applications in various domains where systems interact with incomplete, noisy, or ambiguous data. For instance, in cyber-physical systems, such as autonomous vehicles and drones, runtime monitoring can ensure safety and compliance despite sensor inaccuracies or missing data caused by environmental interference. In healthcare monitoring, systems tracking patient data through medical devices or electronic health records can handle imprecise

or anonymised data, ensuring privacy while providing critical diagnostics. Similarly, in distributed systems and IoT environments, where devices generate out-of-order or incomplete event streams, runtime verification helps maintain system reliability and security. Applications in finance and e-commerce can also benefit, as runtime monitors can verify transactional integrity even under conditions of data encryption or access restrictions.

Expected real-world implications of rationality in runtime verification. The integration of rational monitors with imperfect information in Runtime Verification (RV) offers significant advancements across these domains. For instance, in autonomous robotics, rational monitors dynamically adapt to sensor inaccuracies, enabling drones or underwater vehicles to safely navigate GPS-denied environments. In smart grids, rational monitors maintain stability by making informed adjustments despite fluctuating demands or sensor faults. In healthcare, they interpret noisy or missing physiological data from wearables to provide reliable real-time assessments, crucial for patient safety. By accommodating imperfect information and employing rational decision-making, rational monitors significantly enhance system reliability, adaptability, and safety across these complex, real-world applications.

The paper's structure is as follows. Section 2 reports preliminaries notions. Section 3 introduces our case study. Section 4 formally presents the notion of imperfect information, its implication at the monitoring level and the resulting re-engineering of the standard LTL monitor's synthesis pipeline. Furthermore, Section 5 provides the main tools to introduce monitor rationality. Section 6 reports the details on the prototype that has been developed as a result of the re-engineering process, along with some experiments of its use in a realistic case study. Section 7 positions the paper against the state of the art. Section 8 concludes the paper and discusses some possible future directions.

2 Preliminaries

A system S has an *alphabet* Σ with which it is possible to define the set 2^{Σ} of all its events. Given an alphabet Σ, a finite *trace* $\sigma = ev_0 ev_1 \dots ev_n$, is a finite sequence of n events in 2^{Σ}. Symmetrically, an infinite *trace* $\rho = ev_0 ev_1 \dots$, is an infinite sequence of events in 2^{Σ}. With $\sigma(i)$ or $\sigma[i]$ (resp., $\rho(i)$ or $\rho[i]$), we denote the i-th element of σ (resp., ρ) (*i.e.*, ev_i), $\sigma[i, j]$ the sub-trace starting in i and ending in j, σ^i (resp., ρ^i) the suffix of σ (resp., ρ) starting from i (*i.e.*, $ev_i ev_{i+1} \dots$), $(2^{\Sigma})^*$ the *set of all possible finite traces* over Σ, and $(2^{\Sigma})^{\omega}$ the *set of all possible infinite traces* over Σ. Furthermore, given two traces σ_1 and σ_2 (resp., ρ_2), we denote with $\sigma_1 \bullet \sigma_2$ (resp., $\sigma_1 \bullet \rho_2$) the standard concatenation of the two traces (where the first operand has to be finite while the second one can be infinite). Finally, we denote by $|\sigma|$ the length of σ, which in case of an infinite trace ρ takes value infinity, *i.e.*, $|\rho| = \infty$.

The standard formalism to specify properties in RV is Linear Temporal Logic (LTL [30]). The relevant parts of the syntax of LTL are the following:

$$\varphi := p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \bigcirc\varphi \mid (\varphi \ \mathbf{U} \ \varphi)$$

where $p \in \Sigma$ is an atomic proposition (aka atom), φ is a formula, $\bigcirc$ stands for *next-time*, and $\mathbf{U}$ stands for *until*. In the rest of the paper, we also use the standard derived operators, such as $\top$ instead of $(p \vee \neg p)$, $\bot$ instead of $(p \wedge \neg p)$, $(\varphi \rightarrow \varphi')$ instead of $(\neg\varphi \vee \varphi')$, $\varphi \ R \ \varphi'$ (φ *releases* φ') instead of $\neg(\neg\varphi \, \mathbf{U} \, \neg\varphi')$, $\Box\varphi$ (*always* φ) instead of $(\bot \ R \ \varphi)$, and $\Diamond\varphi$ (*eventually* φ) instead of $(\top \, \mathbf{U} \, \varphi)$.

Let $\rho \in (2^{\Sigma})^{\omega}$ be an infinite sequence of events over Σ, the semantics of LTL is as follows:

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

An infinite trace ρ satisfies an atomic proposition p, if p belongs to $\rho(0)$; which means, p holds in the initial event of the trace ρ. A trace ρ satisfies the negation of the LTL property φ, if ρ does not satisfy φ. A trace ρ satisfies the disjunction of two LTL properties, if ρ satisfies at least one of them. A trace ρ satisfies next-time φ, if the suffix of ρ starting in the next step (ρ^1) satisfies φ. Finally, a trace ρ satisfies $\varphi \,\mathbf{U}\, \varphi'$, if there exists a suffix of ρ 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 $[\![\varphi]\!] = \{\rho \mid \rho \models \varphi\}$.

In Definition 2.1, we present a general and formalism-agnostic definition of a monitor. Informally, 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 2.1 (Monitor). Let S be a system with alphabet Σ, σ a finite trace, and φ be an LTL property. Then, a monitor for φ is a function $Mon_{\varphi} : (2^{\Sigma})^* \to \mathbb{B}_3$, where $\mathbb{B}_3 = \{\top, \bot, ?\}$:

$$Mon_{\varphi}(\sigma) = \begin{cases} \top & \forall_{\rho \in (2^{\Sigma})^{\omega}}.\sigma \bullet \rho \in [\![\varphi]\!] \\ \bot & \forall_{\rho \in (2^{\Sigma})^{\omega}}.\sigma \bullet \rho \notin [\![\varphi]\!] \\ ? & otherwise \end{cases}$$

Intuitively, a monitor returns $\top$ if all continuations (ρ) 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 future events may still change the verdict. 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 a 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) [9, 10]. 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 [10], 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).

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 NBA A^{φ} is generated in step *(ii)*. This can be obtained using Gerth *et al.*'s algorithm [22]. Such automaton recognises the set of infinite traces that satisfy φ (according to LTL semantics). Then, each state of A^{φ} is

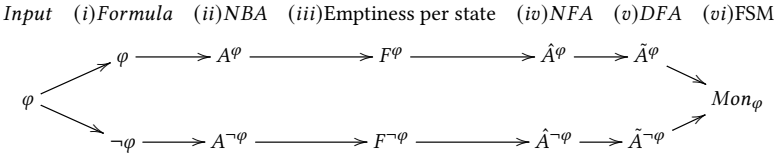


Fig. 1. Steps required to generate an FSM from an LTL formula φ. NBA is Non-deterministic Büchi Automaton [2], NFA is Non-deterministic Finite Automaton [23], and DFA is Deterministic Finite Automaton [23].

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, an NFA $\hat{A}^{\varphi}$ 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 have 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 [31]) into its equivalent deterministic version $\tilde{A}^{\varphi}$ in step *(v)*; this is possible since deterministic and non-deterministic finite 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. This brings us to the revised monitor construction as follows.

Definition 2.2 (Monitor). Given an LTL formula φ and a finite trace σ, the revised monitor is defined as follows:

$$Mon_{\varphi}(\sigma) = \begin{cases} \top & \sigma \notin \mathcal{L}(\tilde{A}^{\neg\varphi}) \\ \bot & \sigma \notin \mathcal{L}(\tilde{A}^{\varphi}) \\ ? & \sigma \in \mathcal{L}(\tilde{A}^{\varphi}) \wedge \sigma \in \mathcal{L}(\tilde{A}^{\neg\varphi}) \end{cases}$$

where $\mathcal{L}(A)$ denotes the language recognised by automaton A.

Now that all the background notions have been introduced, we can move on with the more technical parts of the paper. Specifically, in Section 4, we present how to extend the notion of a monitor in case of imperfect information and, in Section 5, we show how to make a monitor rational in order to reason upon its lack of information. To facilitate the understanding of our technical contribution, we present our case study in the next section, which will serve as a running example in the rest of the paper.

Remark 2.1. *Runtime verification approaches can be broadly categorised as online or offline, with key differences in their execution and application contexts. Online RV operates concurrently with the system, analysing events as they occur to provide real-time monitoring or enforcement. In contrast, offline RV, which is the focus of this work, performs verification after the system has executed, analysing complete traces to ensure comprehensive assessment. This offline setting allows for more thorough and precise evaluations, making it particularly suited to applications where post-execution trace analysis is sufficient or desirable.*

3 Remote inspection case study

Our case study is based on a 3D simulation of a Jackal[1], a four-wheeled unmanned ground vehicle (referred to as the 'rover' from now on), coupled with a simulated radiation sensor, that the rover uses to take radiation readings of points of interest while patrolling around a nuclear facility, and a camera, that the rover uses to inspect images of the nuclear waste barrels in the area. This simulation is based on the work presented in [37], which explains how the simulated sensor works and how radiation was simulated in the environment. In our version of the simulation the rover is autonomously controlled by a rational/intelligent agent [36]. Figure 2 reports a screenshot of the case study.

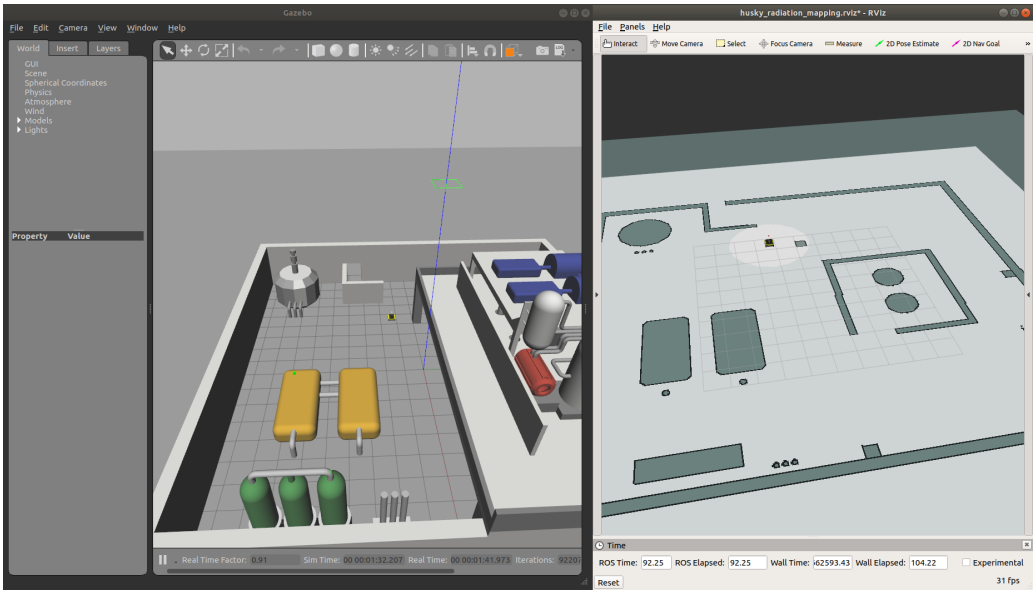


Fig. 2. Simulation in Gazebo [37] of the remote inspection of nuclear plant.

A typical mission in our simulation starts with the rover positioned at the entrance of a nuclear facility. The goal of this mission is to inspect a number of points of interest (*i.e.*, waypoints). Inspecting a waypoint serves two purposes: taking radiation readings to check if the radiation is at an acceptable level, and using a camera to detect abnormalities such as leakage in barrels and pipes. After inspecting all of the waypoints, the rover can either return to the entrance to await for a new mission, or keep patrolling and inspecting the waypoints.

Without losing generality, we assume the image captured by the rover's camera can be represented as a grid. Each cell in such a grid can contain, or not, an abnormality (e.g., a cut in the barrel). This information is translated into propositions, that can be transmitted to the monitor to be analysed at runtime.

We assume the presence of a cut on a barrel, denoted by c, and the presence of a rust stain, denoted by s. Furthermore, we assume the presence of α, β, and γ radiations. The decision of moving back to base for decontamination is denoted by mb, and signalling a warning when a cut is found is denoted by w. To simplify the case study, we assume that the rover only needs to analyse three ordered barrels: b^1, b^2, and b^3. In more detail, when the atomic proposition b^i holds, this means that the barrel is under analysis. Note that, the pure analysis of a barrel does not imply anything on its status,

[1] https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle

the latter is implied by the presence (or not) of the atomic proposition c. Therefore, the set of atomic propositions is $\Sigma = \{b^1, b^2, b^3, c, s, \alpha, \beta, \gamma, mb, w\}$. Furthermore, as is common in RV, the alphabet serves as an abstraction of the actual system events. This implies the assumption of an instrumentation step, where the system – such as the rover – is appropriately equipped with mechanisms to collect and translate low-level events into their corresponding high-level representations. This can be easily obtained exploiting existing frameworks, such as [17].

The first property we want to verify at runtime is that when the rover finds at least a cut (c) on a barrel, it signals the presence of the cut through a warning (w)[2].This property can be formulated as the following LTL formula:

$$\varphi_1 = \Diamond(c \wedge \bigcirc w)$$

Furthermore, we are interested in checking whether the rover behaves properly in the presence of high levels of γ radiation, consequently aborting the mission and going back to base for decontamination (mb). This formula can be formalised as:

$$\varphi_2 = \Diamond(\gamma \wedge (b^1 \vee b^2 \vee b^3) \wedge \bigcirc mb)$$

Last, but not least, we have a property to check that when the rover inspects a barrel without finding a cut, at a certain point it continues the inspection to the next barrel. The formula is formalised as follows:

$$\varphi_3 = \Diamond((\neg c \wedge b^1 \wedge \bigcirc b^2) \vee (\neg c \wedge b^2 \wedge \bigcirc b^3))$$

In addition, we have three safety objectives. The first concerns whether the rover will not find a cut in the barrel. This property can be formulated as the following LTL formula:

$$\psi_1 = \Box((b^1 \vee b^2 \vee b^3) \rightarrow \neg c)$$

Furthermore, another safety property of interest is to check that no γ radiation is detected close to any barrel. This property can be formalised as:

$$\psi_2 = \Box(\gamma \rightarrow \neg(b^1 \vee b^2 \vee b^3))$$

Last, but not least, we have a property to verify that in absence of γ radiation the mission is not aborted. The formula follows:

$$\psi_3 = \Box(\neg\gamma \rightarrow \neg mb)$$

To better understand the properties, let us just assume that the trace of events σ observed by the rover is

$$\sigma(0) = \{\}, \sigma(1) = \{b^1\}, \sigma(2) = \{mb, b^2\}, \sigma(3) = \{\}, \sigma(4) = \{w\}$$

Note that, when applied to σ we have that φ_1 is determined as ? by the standard LTL monitor. In fact, no events in σ contain c. Similarly, in φ_2, we once more obtain ? because no events in σ contains γ. Finally, φ_3 is instead determined as $\top$ by the monitor. In fact, in $\sigma(1)$, b^1 holds, c does not hold, and in $\sigma(2)$, b^2 holds. On the safety properties side, ψ_1 is

[2]Note that, contrary to common understanding, the property is intended to denote a reachability formula – that is, a temporal goal that may or may not be satisfied at runtime. Unlike standard liveness properties, the monitored system is not required to satisfy this property in every run. Rather, the atomic proposition c is used as a trigger: once satisfied, it alerts the rover and initiates the corresponding behaviour. This reasoning can be applied to the other properties of the same class.

determined as ?. This can be derived by the fact that in every event the property holds. Concerning ψ_2, again the result is ?, this is because the left operand of the implication never holds. Finally, ψ_3 is determined as $\bot$ by the monitor. In fact, event $\sigma(2)$ contains mb, but not γ.

4 Runtime verification with imperfect information

So far, we have focused on standard RV of LTL properties. However, this standard approach relies on a critical assumption:

The absence of an atomic proposition is equivalent to
the negation of that proposition.

This assumption holds true in formal verification for systems with perfect information – that is, systems where every component has complete knowledge and visibility of the entire system. While this may be applicable to monolithic and traditional systems, it does not necessarily apply to autonomous, distributed, or artificial intelligence-driven systems. In such scenarios, having a complete view of the system is often not feasible.

Since RV relies on monitoring the system under analysis, if the verified component does not have complete access to the system's information, the monitor will also lack complete access. As a result, our runtime monitor may only observe partial information about the system. Consequently, the event traces provided to the monitor might be missing some atomic propositions, which could be incorrectly interpreted as the negation of those propositions. It is crucial to differentiate between knowing that something is not true and recognising that something is simply unknown.

4.1 How can we formally represent the imperfect information?

As previously discussed in this paper and in [18], the issue with using LTL in systems with imperfect information lies in confusing the absence of an atomic proposition with its negation. When information is imperfect, the trace may lack certain atomic propositions that are simply unknown or unobservable. To address this, we need a method to explicitly characterise the absence of information. To achieve this, we adopt an approach similar to that in [11, 12], where atomic propositions are duplicated.

One possible way to represent imperfect information is by allowing indistinguishability on atomic propositions Σ. To do this we introduce an equivalence relation $\sim$ over Σ and its equivalence classes.

Definition 4.1 (Equivalence relation). An equivalence relation $\sim \in \Sigma \times \Sigma$ determines what a monitor cannot distinguish. Specifically, given two atomic propositions $p, q \in \Sigma$, we say that they are indistinguishable if and only if $p \sim q$.

Definition 4.2 (Equivalence class). Given an equivalence relation $\sim$ we define with $\lambda \in 2^{\Sigma}$ an equivalence class of $\sim$. Formally, given $p \in \lambda$, for each $q \in \Sigma$, if $p \sim q$ then $q \in \lambda$. Additionally, we define the witness of λ with the symbol $[\lambda]$ and with Λ the set of equivalence classes.

To handle the verification process in the imperfect information context, we need to do some extensions. First of all, we can not simply use the set of atomic propositions Σ. In particular, we need to replace Σ with a new set $\bar{\Sigma}$ that is defined as follows: for each $p \in \Sigma$ we have $p_\top \in \bar{\Sigma}$ and $p_\bot \in \bar{\Sigma}$. That is, we duplicate the set of atomic proposition to make the truth value explicit.

Without losing generality, we only consider LTL formulas in Negation Normal Form (NNF)[3]. An LTL formula in NNF has only negations at the atom levels (*i.e.*, we only have $\neg p$). Given an LTL formula, its NNF can be easily obtained

[3]We need to use LTL in NNF to push negation down to the atomic proposition level. This is necessary to define an LTL three-valued semantics and explicitly determine the truth value of atomic propositions.

by propagating all negations to the atoms. For instance, if we had $\neg\bigcirc p$, we would rewrite it as $\bigcirc\neg p$. The same goes for the other operators[4].

Now, we present how to generate the explicit version of an LTL formula.

Definition 4.3. Given an LTL formula φ in NNF and the set of equivalence classes Λ, we define the explicit version of φ as follows:

$$\begin{aligned}
\epsilon(p) &= [\lambda]_\top \\
\epsilon(\neg p) &= [\lambda]_\bot \\
\epsilon(\varphi \vee \varphi') &= \epsilon(\varphi) \vee \epsilon(\varphi') \\
\epsilon(\varphi \wedge \varphi') &= \epsilon(\varphi) \wedge \epsilon(\varphi') \\
\epsilon(\bigcirc\varphi) &= \bigcirc\epsilon(\varphi) \\
\epsilon(\varphi \,\mathbf{U}\, \varphi') &= \epsilon(\varphi) \,\mathbf{U}\, \epsilon(\varphi') \\
\epsilon(\varphi \,\mathbf{R}\, \varphi') &= \epsilon(\varphi) \,\mathbf{R}\, \epsilon(\varphi')
\end{aligned}$$

where $\lambda \in \Lambda$ and $p \in \lambda$.

We now present how to construct the explicit and visible versions of a trace.

Definition 4.4. Given a trace σ and a set Σ, we define the explicit version of σ as σ_e, for each element $\sigma(i)$ as follows:

- for all $p \in \sigma(i)$, $p_\top \in \sigma_e(i)$;
- for all $p \in \Sigma \setminus \sigma(i)$, $p_\bot \in \sigma_e(i)$.

Definition 4.5. Given an explicit trace σ_e and the set of equivalence classes Λ, we define the visible version of σ_e as σ_v, for each $\sigma(i)$ and $\lambda \in \Lambda$ as follows:

- $[\lambda]_\top \in \sigma_v(i)$ if and only if for all $p \in \lambda$, $p_\top \in \sigma_e(i)$;
- $[\lambda]_\bot \in \sigma_v(i)$ if and only if for all $p \in \lambda$, $p_\bot \in \sigma_e(i)$.

For simplicity, in the rest of the paper, in the trivial case where an equivalence class consists of a single atomic proposition, we omit the square brackets.

[4]Notice that, by assuming LTL formulas in NNF, in order to maintain the same expressiveness, we need to include, in the LTL syntax, the dual of each Boolean/temporal operator.

Given the above elements, we define a three-valued semantics $\models_3$ for LTL:

$$
\begin{aligned}
&(\rho \models_3 p) = \top \text{ if } p_\top \in \rho(0)\\
&(\rho \models_3 p) = \bot \text{ if } p_\bot \in \rho(0)\\
&(\rho \models_3 \neg p) = \top \text{ if } p_\bot \in \rho(0)\\
&(\rho \models_3 \neg p) = \bot \text{ if } p_\top \in \rho(0)\\
&(\rho \models_3 \varphi \vee \varphi') = \top \text{ if } (\rho \models_3 \varphi) = \top \text{ or } (\rho \models_3 \varphi') = \top\\
&(\rho \models_3 \varphi \vee \varphi') = \bot \text{ if } (\rho \models_3 \varphi) = \bot \text{ and } (\rho \models_3 \varphi') = \bot\\
&(\rho \models_3 \varphi \wedge \varphi') = \top \text{ if } (\rho \models_3 \varphi) = \top \text{ and } (\rho \models_3 \varphi') = \top\\
&(\rho \models_3 \varphi \wedge \varphi') = \bot \text{ if } (\rho \models_3 \varphi) = \bot \text{ or } (\rho \models_3 \varphi') = \bot\\
&(\rho \models_3 \bigcirc\varphi) = \top \text{ if } (\rho^1 \models_3 \varphi) = \top\\
&(\rho \models_3 \bigcirc\varphi) = \bot \text{ if } (\rho^1 \models_3 \varphi) = \bot\\
&(\rho \models_3 \varphi \,\mathbf{U}\, \varphi') = \top \text{ if } \exists_{i\geq 0}.(\rho^i \models_3 \varphi') = \top \text{ and } \forall_{0\leq j<i}.(\rho^j \models_3 \varphi) = \top\\
&(\rho \models_3 \varphi \,\mathbf{U}\, \varphi') = \bot \text{ if } \forall_{i\geq 0}.(\rho^i \models_3 \varphi') = \bot \text{ or } \exists_{0\leq j<i}.(\rho^j \models_3 \varphi) = \bot\\
&(\rho \models_3 \varphi \,\mathbf{R}\, \varphi') = \top \text{ if } \forall_{j\geq 0}.(\rho^j \models_3 \varphi') = \top \text{ or } (\exists_{i\geq 0}.(\rho^i \models_3 \varphi) = \top \text{ and } \forall_{0\leq j\leq i}.(\rho^j \models_3 \varphi') = \top)\\
&(\rho \models_3 \varphi \,\mathbf{R}\, \varphi') = \bot \text{ if } \exists_{j\geq 0}.(\rho^j \models_3 \varphi') = \bot \text{ and } (\forall_{i\geq 0}.(\rho^i \models_3 \varphi) = \bot \text{ or } \exists_{0\leq j\leq i}.(\rho^j \models_3 \varphi') = \bot)
\end{aligned}
$$

In all the other cases the truth value is undefined (*uu*).

To help the reader, we conclude the section with the following example.

Example 4.6. Considering the rover example presented in Section 3, let us assume that, the rover is not always capable of sending and detecting all the information to the monitor. Because of this, the monitor is sometimes unable to distinguish between a cut and a rust stain, as well as between different kinds of radiation. In such cases, from the viewpoint of the monitor analysing the scene, there is imperfect information over the atomic propositions. Then, we have imperfect information over c and s, which is formalised as $c \sim s$ (*i.e.*, there is an equivalence class λ_{cs} between c and s), and over α, β, and γ, which is formalised as $\alpha \sim \beta \sim \gamma$ (*i.e.*, there is an equivalence class $\lambda_{\alpha\beta\gamma}$ between α, β, and γ). Since in this scenario we have an equivalence relation between c and s (*i.e.*, $c \sim s$) and between α, β, and γ (*i.e.*, $\alpha \sim \beta \sim \gamma$), first we need to make explicit the atomic propositions inside the formula, obtaining:

$$
\begin{aligned}
\epsilon(\varphi_1) &= \Diamond([\lambda_{cs}]_\top \wedge \bigcirc w_\top)\\
\epsilon(\varphi_2) &= \Diamond([\lambda_{\alpha\beta\gamma}]_\top \wedge (b^1_\top \vee b^2_\top \vee b^3_\top) \wedge \bigcirc mb_\top)\\
\epsilon(\varphi_3) &= \Diamond(([\lambda_{cs}]_\bot \wedge b^1_\top \wedge \bigcirc b^2_\top) \vee ([\lambda_{cs}]_\bot \wedge b^2_\top \wedge \bigcirc b^3_\top))\\
\epsilon(\psi_1) &= \Box((b^1_\top \vee b^2_\top \vee b^3_\top) \rightarrow [\lambda_{cs}]_\bot)\\
\epsilon(\psi_2) &= \Box([\lambda_{\alpha\beta\gamma}]_\top \rightarrow (b^1_\bot \wedge b^2_\bot \wedge b^3_\bot))\\
\epsilon(\psi_3) &= \Box([\lambda_{\alpha\beta\gamma}]_\bot \rightarrow mb_\bot)
\end{aligned}
$$

Note that, in Section 3, we assumed, as usual in RV, that the monitor had perfect information over the system. Thus, the trace σ there presented was assumed to perfectly denote the actual events generated by the system execution and observed by the monitor. However, that could not be the case in general, since as we mentioned before, the rover may

not have the ability to send/detect all the information to the monitor, causing imperfect information in the latter. Let us now assume that the trace σ with perfect information was:

$$\sigma(0) = \{\}, \sigma(1) = \{\gamma, b^1, c\}, \sigma(2) = \{\gamma, c, mb, b^2\}, \sigma(3) = \{c\}, \sigma(4) = \{w\}$$

Thus, we can update the trace of events of Section 3 as well, first by generating its explicit version σ_e (see Definition 4.4), where

$$\begin{aligned}
\sigma_e(0) &= \{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\bot\}\\
\sigma_e(1) &= \{b^1_\top, b^2_\bot, b^3_\bot, c_\top, s_\bot, \alpha_\bot, \beta_\bot, \gamma_\top, mb_\bot, w_\bot\}\\
\sigma_e(2) &= \{b^1_\bot, b^2_\top, b^3_\bot, c_\top, s_\bot, \alpha_\bot, \beta_\bot, \gamma_\top, mb_\top, w_\bot\}\\
\sigma_e(3) &= \{b^1_\bot, b^2_\bot, b^3_\bot, c_\top, s_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\bot\}\\
\sigma_e(4) &= \{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\top\}
\end{aligned}$$

Then by defining its visible version according to the given equivalence classes λ_{cs} and $\lambda_{\alpha\beta\gamma}$ (see Definition 4.5), we obtain σ_v, where

$$\begin{aligned}
\sigma_v(0) &= \{b^1_\bot, b^2_\bot, b^3_\bot, [\lambda_{cs}]_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}\\
\sigma_v(1) &= \{b^1_\top, b^2_\bot, b^3_\bot, mb_\bot, w_\bot\}\\
\sigma_v(2) &= \{b^1_\bot, b^2_\top, b^3_\bot, mb_\top, w_\bot\}\\
\sigma_v(3) &= \{b^1_\bot, b^2_\bot, b^3_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}\\
\sigma_v(4) &= \{b^1_\bot, b^2_\bot, b^3_\bot, [\lambda_{cs}]_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\top\}
\end{aligned}$$

Note that, as expected, the second and third events in σ_v do not contain information about the atomic propositions c and γ. Furthermore, there is no information on the atomic proposition c in the fourth event. This is determined by the fact that the atomic propositions $c_\top$ and $s_\bot$ hold in the second, third, and fourth events of σ_v, and according to Definition 4.5, since $c \sim s$, we can have $[\lambda_{cs}]_\top$ (resp., $[\lambda_{cs}]_\bot$) if and only if both $c_\top$ and $s_\top$ hold (resp., $c_\bot$ and $s_\bot$). Thus, having a mismatch between the two atomic propositions (*i.e.*, one is true while the other is false), we cannot safely add any witness for the equivalence class λ_{cs}. Instead, in the first and last events of σ_v, since we have both $c_\bot$ and $s_\bot$, we can safely add the witness $[\lambda_{cs}]_\bot$ to the trace. The same reasoning follows for the atomic proposition γ and its equivalence class.

4.2 Re-engineering the monitor with imperfect information

Given an LTL formula and a visible trace for the monitor, we need a method to use them for performing RV. This can be achieved by extending the standard pipeline for generating LTL monitors (see Figure 1). This extension involves two specific modifications:

(1) We use the explicit version of the LTL formula, following Definition 4.3.
(2) We modify the product between $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ to generate the Moore machine representing the monitor.

The resulting extension is illustrated in Figure 3. In this figure, the explicit version of the LTL formula is generated in step (ii), and the updated product between the automata is obtained in step (vii). The remaining steps are unchanged compared to Figure 1 with the exception that the atomic propositions in the formula are duplicated before using the formula to generate the corresponding NBA, and an additional automaton has been added. This duplication of atomic propositions is crucial, as it completely changes the semantics of the subsequent steps in the monitor synthesis

pipeline. Specifically, it is not true that for any given visible trace σ_v, we have $\sigma_v \notin \mathcal{L}(\hat{A}^{\varphi}) \Rightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\neg\varphi})$, nor $\sigma_v \notin \mathcal{L}(\hat{A}^{\neg\varphi}) \Rightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\varphi})$. This means that when a visible trace of events σ_v is not a good prefix for φ (*i.e.*, a prefix that can be extended to an infinite trace satisfying φ), it does not necessarily have to be a bad prefix for φ (*i.e.*, a prefix that cannot be extended to an infinite trace satisfying φ). This aspect is closely related to the reason why a third formula ($\otimes\varphi$) has been introduced in Figure 3. Since duplicating the atomic propositions in the formula breaks the duality between φ and $\neg\varphi$, a third automaton ($\tilde{A}^{\otimes\varphi}$) is needed to recognise all the traces that neither satisfy nor violate φ. For this reason, we extended the pipeline by adding $\otimes\varphi$, which is an abbreviation for $\neg\epsilon(\varphi) \wedge \neg\epsilon(\neg\varphi)$. The automaton $\tilde{A}^{\otimes\varphi}$, obtained by following the same steps as for the positive $\tilde{A}^{\epsilon(\varphi)}$ and negative $\tilde{A}^{\epsilon(\neg\varphi)}$ automata, recognises all prefixes for which no continuation satisfying or violating φ exists.

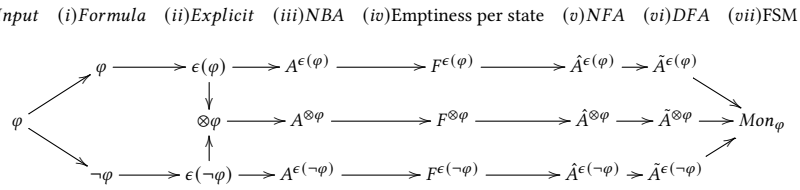


Fig. 3. Extended pipeline to consider imperfect information.

Now, we formalise the above reasoning with the following lemma.

Lemma 4.7. *Given a visible finite trace σ_v and an LTL formula φ, we have:*

$$\sigma_v \notin \mathcal{L}(\hat{A}^{\epsilon(\varphi)}) \nRightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\neg\varphi)})$$
$$\sigma_v \notin \mathcal{L}(\hat{A}^{\epsilon(\neg\varphi)}) \nRightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\varphi)})$$

Proof. Assume we have a visible trace σ_v that is not included in the NFA $\hat{A}^{\epsilon(\varphi)}$. To prove our result, we need to show that σ_v is also not included in $\hat{A}^{\epsilon(\neg\varphi)}$. To demonstrate this, suppose $\Sigma = \{p, q, r\}$, $\varphi = \bigcirc p$, $p \sim q$, and σ where $\sigma(1) = \{p\}$. Given Definitions 4.4 and 4.5, we can conclude that $\sigma_v(1) = \{r_\perp\}$. Therefore, σ_v does not satisfy φ and, consequently, is not included in the NFA $\hat{A}^{\epsilon(\varphi)}$. However, it is also not included in $\hat{A}^{\epsilon(\neg\varphi)}$. This is because $p_\top$ and $p_\perp$ are not included in $\sigma_v(1)$. This concludes the first relation. For the second relation, we can use a variant of the above reasoning. □

By adding the third automaton, the corresponding FSM synthesis also needs to change. The revised version is detailed in the following definition.

Definition 4.8 (Monitor with imperfect information). Given an LTL formula φ and a visible finite trace σ_v, a monitor with imperfect information is so defined:

$$Mon^v_\varphi(\sigma_v) = \begin{cases} \top & \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\otimes\varphi}) \\ \bot & \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\otimes\varphi}) \\ uu & \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi}) \\ ?_{\not\bot} & \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi}) \\ ?_{\not\top} & \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi}) \\ ? & \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi}) \end{cases}$$

In this definition, we see how the inclusion of a third automaton in the equation allows us to synthesise a finer monitor, in terms of the number of possible outcomes it returns. Compared to Definition 2.2, we now have three additional outcomes. Specifically, given a visible trace σ_v, the monitor returns $\top$ if there is no continuation of σ_v that either violates $\epsilon(\varphi)$ or makes it undefined. Conversely, it returns $\bot$ if there is no continuation that either satisfies $\epsilon(\varphi)$ or makes it undefined. With three automata, there is an additional final outcome to consider, which is *uu*. Thus, the monitor returns *uu* if there is no continuation that either satisfies or violates $\epsilon(\varphi)$. These first three outcomes derive from the three-valued semantics for LTL. Additionally, we may encounter $?_{\not\bot}$, which is read as "unknown, but it will never be violated from the monitor's point of view". This outcome is returned when the visible trace σ_v has no continuation that will eventually violate $\epsilon(\varphi)$, but there are continuations that satisfy $\epsilon(\varphi)$ and make it undefined. Symmetrically, we have $?_{\not\top}$, which is read as "unknown, but it will never be satisfied from the monitor's point of view". This outcome is the dual of the previous one, where no continuations satisfying $\epsilon(\varphi)$ can be found, but continuations that violate $\epsilon(\varphi)$ and make it undefined exist. Lastly, we may encounter ?, denoting the completely unknown case. This outcome concerns situations where the monitor cannot yet conclude anything, as there exist continuations satisfying $\epsilon(\varphi)$, continuations violating $\epsilon(\varphi)$, and continuations that make it undefined.

Remark 4.1. *Note that, in Definition 4.8, not all possible combinations are included. Specifically, it is not possible to have* $\sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\otimes\varphi})$ *and* $\sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\varphi)}) \wedge \sigma_v \in \mathcal{L}(\tilde{A}^{\epsilon(\neg\varphi)}) \wedge \sigma_v \notin \mathcal{L}(\tilde{A}^{\otimes\varphi})$*. The former is not possible because, by the definition of the three-valued semantics for LTL, there exists at least one automaton that includes the trace. The latter follows from the fact that it is unfeasible, given the nature of a visible trace, for a formula to be both true and false but not undefined in the future.*

In what follows, we provide two preservation results from the monitor with imperfect information to the one with perfect information.

Lemma 4.9. *Given a finite trace σ, a monitor with its visibility $Mon^v_\varphi(\sigma)$, and a general monitor $Mon_\varphi(\sigma)$, we have that:*

$$\text{if } Mon^v_\varphi(\sigma_v) = \top \text{ then } Mon_\varphi(\sigma) = \top \tag{1}$$

$$\text{if } Mon^v_\varphi(\sigma_v) = \bot \text{ then } Mon_\varphi(\sigma) = \bot \tag{2}$$

Proof.

(1) Suppose $Mon^v_\varphi(\sigma_v) = \top$. This means that the visible trace σ_v satisfies the formula $\epsilon(\varphi)$. We want to prove that the original trace σ satisfies the formula φ. To do this, given σ_v, by Definitions 4.4 and 4.5, we know that for each $\sigma_v(i)$, for

all $p_\top \in \sigma_v(i)$, $p \in \sigma(i)$, and for all $p_\bot \in \sigma_v(i)$, $p \notin \sigma(i)$. Given the above reasoning, we need to provide an induction proof over the structure of the formula $\epsilon(\varphi)$.

Case: $\epsilon(\varphi) = p_\top$. So, $\varphi = p$. By hypothesis, $Mon^v_\varphi(\sigma_v) = \top$. By the semantics of three-valued LTL, this means that $p_\top \in \sigma_v(0)$, and by Definitions 4.4 and 4.5, $p \in \sigma(0)$. Therefore, $Mon_\varphi(\sigma) = \top$.

Case: $\epsilon(\varphi) = p_\bot$. Thus, $\varphi = \neg p$. By hypothesis, $Mon^v_\varphi(\sigma_v) = \top$. By the semantics of three-valued LTL, this means that $p_\bot \in \sigma_v(0)$, and by Definitions 4.4 and 4.5, $p \notin \sigma(0)$. Therefore, $Mon_\varphi(\sigma) = \top$.

Since in the inductive cases the transformation of Definition 4.3 does not change the structure and elements of the formula, we can conclude the proof.

(2) Suppose $Mon^v_\varphi(\sigma_v) = \bot$. This means that the visible trace σ_v does not satisfy the formula $\epsilon(\varphi)$. We want to prove that the original trace σ does the same for the formula φ. As in the previous case, we need to prove the implication by induction over the structure of the formula $\epsilon(\varphi)$ for the base cases.

Case: $\epsilon(\varphi) = p_\top$. So, $\varphi = p$. By hypothesis, $Mon^v_\varphi(\sigma_v) = \bot$. By the semantics of three-valued LTL, this means that $p_\bot \in \sigma_v(0)$, and by Definitions 4.4 and 4.5, $p \notin \sigma(0)$. Therefore, $Mon_\varphi(\sigma) = \bot$.

Case: $\epsilon(\varphi) = p_\bot$. Thus, $\varphi = \neg p$. By hypothesis, $Mon^v_\varphi(\sigma_v) = \bot$. By the semantics of three-valued LTL, this means that $p_\top \in \sigma_v(0)$, and by Definitions 4.4 and 4.5, $p \in \sigma(0)$. Therefore, $Mon_\varphi(\sigma) = \bot$.

□

Given the above results, we can deduce the following corollary.

Corollary 4.10. *Given a visible finite trace σ_v and an LTL formula φ, we have:*

$$\sigma_v \notin \mathcal{L}(\hat{A}^{\epsilon(\varphi)}) \Rightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\neg\varphi)}) \vee \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi})$$
$$\sigma_v \notin \mathcal{L}(\hat{A}^{\epsilon(\neg\varphi)}) \Rightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\varphi)}) \vee \sigma_v \in \mathcal{L}(\tilde{A}^{\otimes\varphi})$$
$$\sigma_v \notin \mathcal{L}(\tilde{A}^{\otimes\varphi}) \Rightarrow \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\neg\varphi)}) \vee \sigma_v \in \mathcal{L}(\hat{A}^{\epsilon(\varphi)})$$

Example 4.11. Considering once more the running example presented in Section 3 and Example 4.6. Thanks to our three-value semantics and the presence of explicit atomic propositions, the trace σ which was erroneously classifying φ_3 as $\top$ and ψ_3 as $\bot$ from the standard LTL monitor before, now is classified as $?_{\not\perp}$ and $?_{\not\top}$, respectively. The semantics of the two verdicts is fundamentally different, as well as the reaction that the system should have. In the first case, by using a standard LTL monitor, the verdict returned by the monitor was $\top$. Thus, the agent controlling the rover could have used such information to continue the inspection with another barrel and not detecting a danger. In the second case, by using the extended LTL monitor that we presented in this work, the verdict returned by the monitor was $?_{\not\perp}$. Thus, the agent controlling the rover could use this information to, for instance, ask the rover to check again, maybe taking another picture. Even though this is a simple example, it allows us to show how our extension tackles the foundations of the imperfect information issue. A complementary reasoning can be followed for ψ_3. The rest of the properties are not reported because the extended LTL monitor keeps returning an inconclusive verdict as the standard LTL monitor.

5 Runtime verification with rational monitor

In Section 4, we demonstrated how to engineer a monitor with imperfect information and specified it by extending the classical pipeline for creating a standard three-valued monitor. Note that, through this contribution, we enable the monitor to avoid incorrect truth values, ensuring the correctness of the truth value returned by the monitor. However,

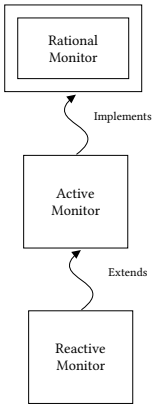


Fig. 4. Rational Monitor hierarchy.

the monitor is "passive" concerning imperfect information. In this section, we will focus on improving the monitor's ability to handle imperfect information, making it "active". To do this, we introduce the concept of resources on the monitor's visibility. Specifically, the monitor is able to select the atoms that it wants to see w.r.t. its limited resources. We will introduce in detail this notion in Section 5.1. We further analyse the monitor's ability by introducing in Section 5.2 the notion of reactive monitor. To achieve this, we assume that the monitor can dynamically modify its visibility in different time windows. In Figure 4, we provide a high-level overview, with a sort of class diagram, of the relationship between the rational, active, and reactive monitors.

5.1 How can reasoning upon resources help fighting imperfect information?

As pointed out before, we introduce the notion of resources in order to grant the monitor the ability of reasoning upon its own visibility. That is, when we claim that a monitor does not have visibility over a set of atomic propositions, this does not mean the monitor does not have a mean to access such information (in general), but, that by accessing such information it may incur a certain cost.

Let us consider a robotic scenario where a robot operates in an environment and can only access information provided by its sensors. Such information can be incomplete. In fact, the robot may have limited energy consumption capabilities (*i.e.*, it has a resource bound), making it unreasonable to access all sensors at once. This limitation affects how much power the robot can allocate to its various functions, such as accessing and operating its sensors. Consequently, the robot must manage its energy efficiently, which may involve prioritising certain sensors over others, thus causing incomplete information, to conserve energy and ensure longer operational periods.

In the rest of the section, we formalise the notion of resources and costs associated to the monitor.

Given a set of atomic propositions Σ and an equivalence relation $\sim$, we can derive the set of equivalence classes Λ (according to Definition 4.2). Thus, we define $cost\colon \Lambda \to \mathbb{N}$ that assign for each equivalence class $\lambda \in \Lambda$ a natural number $cost(\lambda)$. The latter represents the cost for the monitor to make visible (or break) the atomic propositions involved in the equivalence class λ.

In this work, we assume our monitor has a limited number of resources and, by consequence, it could be unable to break all the equivalence classes.

So, given a cost function and the resource bound of the monitor, the latter needs to determine the best selection of the atomic propositions. To do so, we assume another function $payoff\colon \Lambda \to \mathbb{N}$ that assigns for each $\lambda \in \Lambda$ a natural number that represents the expected payoff of breaking λ.

To generate a payoff, we need to consider the formula φ under examination and the atomic propositions involved in it. To do this, we can define another function $metric_\varphi : \Sigma \to [0,1]$ that assigns a metric $metric_\varphi(p)$ to each atomic proposition. There are different approaches to producing the $payoff$ function. In the rest of the paper, we assume that $payoff(\lambda) = \sum_{p\in\lambda} metric_\varphi(p)$, meaning the payoff of λ is given by the sum of all the metrics of the atomic propositions involved in λ.

To assign a metric for each atomic proposition, we need to consider the relevance of it in terms of the LTL formula under exam. In particular, we first need to assign a value for each syntactic element in the LTL syntax and then study the structure of the formula to determine the corresponding metrics.

Example 5.1. For instance, we can consider the following evaluation:

$$\begin{aligned}
metric_q(p) &= 0 \ with\ p \neq q\\
metric_p(p) &= 1\\
metric_{\neg q}(p) &= 0\\
metric_{\neg p}(p) &= 1\\
metric_{\varphi\vee\varphi'}(p) &= (metric_\varphi(p) + metric_{\varphi'}(p))/2\\
metric_{\varphi\wedge\varphi'}(p) &= max(metric_\varphi(p), metric_{\varphi'}(p))\\
metric_{\varphi\to\varphi'}(p) &= (metric_\varphi(p) + metric_{\varphi'}(p))/2\\
metric_{\bigcirc\varphi}(p) &= 0.5 \times metric_\varphi(p)\\
metric_{\varphi\,\mathbf{U}\,\varphi'}(p) &= 0.3 \times metric_\varphi(p) + 0.7 \times metric_{\varphi'}(p)\\
metric_{\varphi\,\mathbf{R}\,\varphi'}(p) &= 0.3 \times metric_\varphi(p) + 0.7 \times metric_{\varphi'}(p)
\end{aligned}$$

This metric represents one possible approach, chosen for its intuitive nature, though it is not the only option. Specifically, when analysing atomic propositions, we assign a value of 0 if the proposition under consideration differs from the one specified in the formula, and 1 if they match. The same reasoning applies to the negation of atomic propositions.

For disjunctions (and implications), the metric is calculated as the average value of the subformulas, ensuring that each one is weighted fairly. For conjunctions, however, we adopt a different approach: the metric takes the maximum value among the subformulas. This strategy is justified because the metric reflects the "payoff" of the atomic proposition p. At runtime, the operand with the strongest dependency on p will dictate the satisfaction of the conjunction formula.

For other operators, the metric is designed to align with their temporal semantics. For example, in the case of the $\mathbf{U}$ operator, the metric places greater emphasis on the right operand, as it determines whether the formula is ultimately satisfied.

Given the *cost* function and the $payoff$ function, our aim is to let the monitor able to select the best set of equivalence classes to break. To select such a subset we can use classic dynamic programming approaches like the one of the knapsack problem in which it is possible to minimise the costs while maximising the metrics.

Algorithm 1 outlines the process for achieving active monitoring; it takes in input a finite trace σ, the property to monitor φ, the indistinguishability relation $\sim$, the payoff function to be used based on the metric, the cost function to

Algorithm 1 ActiveMonitor $\langle \sigma, \varphi, \sim, payoff, cost, \mathbf{b} \rangle$

1: $count = 1$
2: $break = knapsack(payoff, cost, \mathbf{b})$
3: $\sim' = (\sim \setminus break)$
4: $Mon_\varphi = Monitor(\varphi)$
5: **while** $count \leq |\sigma|$ **do**
6: $\quad \sigma[count] = \sigma[count] \setminus \sim'$
7: $\quad count = count + 1$
8: **return** $Mon_\varphi(\sigma[1, count])$

determine how much it costs to break an indistinguishability relation, and a bound **b** that represents the maximum amount of resources for the rational monitor. Initially, the algorithm selects which indistinguishability relations to break (lines 2 and 3), leveraging the well-known knapsack algorithm [14] to balance the trade-off between the cost and the payoff of breaking an indistinguishability relation (line 2). This step can be executed through various methods, with the knapsack algorithm being just one possible optimisation technique. Note that, since the knapsack algorithm is performed on sets of atomic propositions (the equivalence classes), we define an order on the equivalence classes with the same payoff. That is, if two equivalence classes score the same payoff, then the knapsack algorithm chooses to break the equivalence classes with the greater number of atomic propositions. In case the number of atomic propositions is the same, then the choice is random. Subsequently, the algorithm updates the indistinguishability relation based on the knapsack problem's output (line 3)[5].

Next, a monitor for the formula φ is generated (line 4), and the trace σ is updated in accordance with the new indistinguishability relation[6] (lines 5–7). By the end of the loop, σ represents the monitor's visible trace based on its visibility criteria (what we previously referred in the paper as σ_v). Finally, the monitor's outcome is returned (line 8).

Example 5.2. Considering the running example presented in Section 3 and revisited in Example 4.6 and Example 4.11, we now show the impact of an active monitor. Suppose that the cost of breaking the equivalence class λ_{cs} is 2 and the cost of breaking the equivalence class $\lambda_{\alpha\beta\gamma}$ is 3. Given the metric of Example 5.1 and the formula φ_1, we can determine the payoff of the atoms of the equivalence classes λ_{cs} and $\lambda_{\alpha\beta\gamma}$ (*i.e.*, $metric_{\varphi_1}$). In more detail, we obtain that the payoff to break the equivalence class λ_{cs} is 0.7, which can be obtained through the computation: $payoff(\lambda_{cs}) = metric_{\varphi_1}(c) + metric_{\varphi_1}(s)$, with $metric_{\varphi_1}(c) = 0.3 \times 0 + 0.7 \times max(1, 0.5 \times 0) = 0.7$, and $metric_{\varphi_1}(s) = 0.3 \times 0 + 0.7 \times max(0, 0.5 \times 0) = 0.0$. With the same reasoning, we obtain that $payoff(\lambda_{\alpha\beta\gamma}) = 0.0$, since no atomic proposition in $\lambda_{\alpha\beta\gamma}$ is of interest for the verification of φ_1. By assuming a bound **b** greater or equal than 2, the ActiveMonitor can break λ_{cs} and generate a new trace of events:

$$\sigma_v(0) = \{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$$
$$\sigma_v(1) = \{b^1_\top, b^2_\bot, b^3_\bot, c_\top, s_\bot, mb_\bot, w_\bot\}$$
$$\sigma_v(2) = \{b^1_\bot, b^2_\top, b^3_\bot, c_\top, s_\bot, mb_\top, w_\bot\}$$
$$\sigma_v(3) = \{b^1_\bot, b^2_\bot, b^3_\bot, c_\top, s_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$$
$$\sigma_v(4) = \{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, [\lambda_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\top\}$$

[5]Note that, $Set_1 \setminus Set_2$ denotes the classic set operation that removes from Set_1 all the elements in Set_2.
[6]Note that, with $\sigma[count] \setminus \sim'$ we represent the event $\sigma[count]$ in which we remove all atomic propositions involved in $\sim'$. In addition, for each equivalence class in which each atomic proposition has the same truth value, we add the corresponding witness..

Thanks to its ability, the ActiveMonitor can conclude with $\top$ for the liveness property φ_1. This is determined by the fact that in $\sigma_v(3)$ the atomic proposition $c_\top$ holds and in $\sigma_v(4)$ the atomic proposition $w_\top$ holds. The same reasoning can be followed for the liveness property φ_2, in which instead of breaking λ_{cs}, the $\lambda_{\alpha\beta\gamma}$ equivalence class is broken. Note that, to accomplish the task of breaking the latter equivalence class the ActiveMonitor needs to have **b** greater or equal than 3. The same reasoning can be followed to handle the safety formulas ψ_1 and ψ_2 presented in Section 3.

Theorem 5.3. *Algorithm 1 terminates in double exponential-time in the size of the LTL formula φ.*

Proof. We now analyse the key steps of the algorithm to determine its time complexity. First, we examine the complexities of the individual procedures involved. In line 2, a classical knapsack algorithm is used to decide which equivalence classes to break, incurring a pseudo-polynomial cost with respect to the input [26]. Another procedure is the monitor synthesis in line 4, which is well known to be double-exponential in the size of the formula [10]. The while loop in lines 5–7 iterates over the length of the trace σ, resulting in a linear cost relative to σ. Finally, in line 8, the monitor is invoked to verify the property φ on σ, again with a linear cost in terms of the trace length [10]. Taking all these elements into account, the resulting complexity follows. □

Remark 5.1. *As in standard runtime verification, monitor synthesis can be performed prior to system execution, as discussed in [10]. Thus, the operation in line 4 can be computed offline, ensuring that it does not affect runtime performance. Similarly, the knapsack algorithm (line 2) can also be executed beforehand, prior to the active monitor's runtime execution. Thanks to this preprocessing, the actual runtime monitoring complexity can be reduced to linear in the size of the input.*

5.2 How can we formally represent the dynamic behaviour of the monitor?

In the previous section, we have introduced the notion of active monitor. However, our goal is to introduce a monitor able to dynamically reason upon its imperfect information. Thus, in this section, we introduce the notion of "reactive" monitor.

First, we need to introduce the concept of a *time window*. A time window allows us to split the input trace into different frames. We can assume that within each frame, the monitor's visibility is static, while it can change between frames, making it dynamic. At the end of each frame, the monitor can reallocate its resources (*i.e.*, determine which equivalence relations are in place).

For each frame, the monitor can use the approach proposed in Section 4.1, allowing it to adjust the allocation of its resources. However, this alone is not sufficient to make the monitor rational. The presence of time windows does not inherently ensure dynamicity in the monitor's visibility. The parameters that need to change include the formula under examination and the associated payoff. Without a new formula, the monitor would break the same equivalence relations in each time frame.

To update the formula, we propose removing the sub-formulas that have been verified (or violated) during the previous frames. With this new formula, we can generate a new payoff. Using this updated payoff, the monitor can adapt its strategy via the optimisation algorithm, selecting new equivalence relations to break.

Algorithm 2 outlines the steps required to synthesize and apply a reactive monitor. Notably, the structure of Algorithm 2 closely resembles that of Algorithm 1, particularly in the initialisation steps (lines 1–4), where the knapsack algorithm updates the indistinguishability relation and synthesises the monitor for the formula φ. The primary difference lies in the introduction of an if statement (lines 6–13), where the indistinguishability relation ($\sim$) and the trace (σ) are

Algorithm 2 ReactiveMonitor $\langle \sigma, \varphi, \sim, payoff, cost, \mathbf{b}, window \rangle$

```
1:  count = 1
2:  break = knapsack(payoff, cost, b)
3:  ∼′ = (∼ \ break)
4:  Mon_φ = Monitor(φ)
5:  while count ≤ |σ| do
6:      if count mod window = 0 then
7:          if count < window then
8:              update φ according to Mon_φ(σ[1, count])
9:          else
10:             update φ according to Mon_φ(σ[count − window, count])
11:         update payoff according to φ
12:         break = knapsack(payoff, cost, b)
13:         ∼′ = (∼ \ break)
14:     σ[count] = σ[count]\ ∼′
15:     count = count + 1
16: return Mon_φ(σ[1, count])
```

updated. This update occurs whenever the time window (specified as input to the algorithm) expires (checked on line 6). At this point, the reactive monitor updates the formula φ based on past events observed in σ (lines 7–10), reflecting what the monitor has verified considering these past events. Subsequently, the payoff is updated in accordance with the new formula φ (line 11). The knapsack algorithm is then iteratively applied to update the indistinguishability relation (lines 12–13). The rest of the algorithm proceeds in the same manner as Algorithm 1.

Remark 5.2. *Algorithm 2 updates the LTL formula at execution time solely to apply the metric and calculate the new payoffs for breaking equivalence classes, which is necessary due to the metric being defined on the LTL formula's structure. However, the monitor itself does not require updates during the execution of the system. Thus, the resulting Moore machine used for evaluating σ is independent of the current state of $\sim$, as demonstrated in Figure 3 (i.e., $\sim$ is not an input for the monitor synthesis). The updated $\sim$ is only relevant for identifying the appropriate events based on the monitor's current visibility (line 13).*

Example 5.4. Considering one last time the running example presented in Section 3 and Examples 4.6–5.2. Let us assume the monitor needs to verify a combination of the formulas previously introduced. For example, we may consider a new formula $\psi = \psi_1 \vee \psi_2$. Now, if we consider an ActiveMonitor, with the same resource bound as in Example 5.2, we can easily note that the monitor cannot determine a different truth value of the monitor introduced in Section 4. The reason lies in the need of breaking two equivalence classes; however, since the cost of breaking both classes (*i.e.*, 5) is greater than the bound (*i.e.*, 3), the ActiveMonitor has not the ability to conclude. This brings us to the use of a ReactiveMonitor instead, which, differently from the ActiveMonitor counterpart, is capable of selecting which equivalence classes to break dynamically. Suppose that we have a time window of 2, the ReactiveMonitor can break first $\lambda_{\alpha\beta\gamma}$ and by doing so falsifying ψ_2 in $\sigma_v(1)$. Then, the monitor can break λ_{cs} in the second time window (*i.e.*, $\sigma_v(2)$ and $\sigma_v(3)$). By doing so, it can falsify ψ_1 in $\sigma_v(2)$. However, notice that this resolution depends on the chosen payoff. In fact, if the monitor had selected first λ_{cs} and then $\lambda_{\alpha\beta\gamma}$ it would not have concluded the violation of ϕ. In our example, we have the following metrics:

$$metric_{\psi}(c) = (metric_{\psi_1} + metric_{\psi_2})/2 =$$

$$(((0.3 \times 0) + (0.7 \times ((0.0) + (1))/2)) + 0.0)/2 = (0.7 \times 0.5)/2 = 0.175$$

$$metric_{\psi}(s) = 0.0$$
$$metric_{\psi}(\gamma) = (metric_{\psi_1} + metric_{\psi_2})/2 =$$
$$(0.0 + ((0.3 \times 0) + (0.7 \times ((1.0 + 0.0)/2))))/2 = (0.7 \times 0.5)/2 = 0.175$$
$$metric_{\psi}(\alpha) = 0.0$$
$$metric_{\psi}(\beta) = 0.0$$

Note that, even though the payoff of $\lambda_{\alpha\beta\gamma}$ and λ_{cs} are equal, ReactiveMonitor breaks $\lambda_{\alpha\beta\gamma}$ first since it comprises more atomic propositions.

Theorem 5.5. *Algorithm 2 terminates in double exponential-time in the size of the LTL formula φ.*

Proof. We now analyse the key steps of the algorithm to determine its time complexity. As with Algorithm 1, we begin by examining the complexities of the individual procedures involved. In lines 2 and 12, a classical knapsack algorithm is used to determine which equivalence classes should be broken, incurring a pseudo-polynomial cost relative to the input [26]. Another procedure, the monitor synthesis in line 4, is known to be double exponential in the size of the formula [10]. The while loop in lines 5–15 iterates over the length of the trace σ, calling the knapsack algorithm at each iteration and computing the payoff function. Since the payoff function can be computed in linear time with respect to the number of atomic propositions in φ, the complexity of each iteration is dominated by the knapsack algorithm, resulting in a pseudo-polynomial cost overall. Finally, in line 16, the monitor is invoked to verify the property φ on σ, again at a linear cost in terms of the trace length [10]. Considering all these factors, the resulting complexity follows from the combination of these costs. □

Remark 5.3. *As mentioned for Algorithm 1, monitor synthesis can be performed before the system is executed, as discussed in [10]. Thus, the operation in line 4 can be carried out offline, ensuring that it does not impact runtime performance. Similarly, the knapsack algorithm (lines 2 and 12) can also be computed in advance, before the reactive monitor's execution. Note that, to enable this, we assume the computation of a matrix covering all possible payoff values as input to the problem. With these preprocessing steps, the actual runtime monitoring complexity can be reduced to linear in the size of the input.*

To conclude this section, Figure 5 recaps an overview of our approach's pipeline. Beginning with an LTL formula φ, a payoff function is generated (step 1) and used to apply the knapsack algorithm (step 2). Following this, the monitor is synthesised (step 3). If the entire trace σ has been analysed, the monitor can return the outcome (step 4b). If not (step 4a), the formula is revised in accordance with σ (step 5), and a new payoff is generated (step 6), initiating another iteration of the pipeline. Note that when considering an active monitor, steps 4a, 5, and 6 can be omitted since we are dealing with a single time window.

6 Implementation

The prototype implementing the theory discussed in this paper is publicly available on GitHub[7]. It consists of a Python script that implements the entire pipelines illustrated in Figure 3 and Figure 5. Python was chosen for its rich library

[7] https://github.com/AngeloFerrando/RationalMonitor

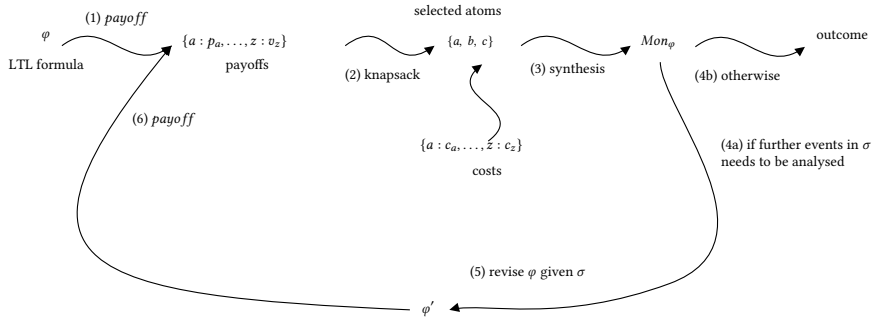


Fig. 5. Overview of the pipeline of the work.

ecosystem, particularly the Spot library[8] [15], which is well-suited for automaton manipulation. Specifically for the monitor synthesis, we utilised Spot to automatically generate a Non-deterministic Büchi Automaton (NBA) from an LTL formula, corresponding to step (iii) in Figure 3, which is the most complex and computationally demanding step in the pipeline. The remainder of the pipeline was directly implemented in Python.

The prototype is encapsulated in a Python class named 'RationalMonitor'. To create an instance of 'RationalMonitor', the constructor requires the following inputs:

(i) An LTL formula to verify;
(ii) A set of atomic propositions;
(iii) One or more equivalence classes over the same set of atomic propositions;
(iv) The metric function to evaluate and assign payoffs to equivalence classes;
(v) The costs associated with breaking these equivalence classes;
(vi) The resource bound for the monitor;
(vii) A time window;
(viii) A trace of events to analyse.

These parameters are related to the inputs required by Algorithm 2, which subsumes those of Algorithm 1. In fact, in case of an active monitor, the time window is not necessary and can be left unspecified.

Using this information, a FSM representing the monitor, as defined in Definition 4.8, is constructed. The monitor then analyses the input trace and returns the corresponding verdict to the user. The trace is typically stored in a file (e.g., a log file). These input parameters can be supplied as command-line arguments to the tool. However, since the monitor is represented as a single data structure, it is also possible—and often practical—to import the script and use the monitor programmatically (*i.e.*, the prototype can be used as a RV library as well). This flexibility is particularly useful for online verification scenarios, in addition to offline verification.

Remark 6.1. *Although Algorithm 1 and Algorithm 2 are presented in an offline scenario, where the complete event trace σ is provided as input (e.g., from a log file), our implementation adopts an incremental approach. In practice, the monitor operates on a growing prefix of the trace, reflecting the ongoing execution of the system. This incremental adaptation does not alter the core logic of our algorithms; it is a minimal change made to align with practical use. We presented the offline*

[8] https://spot.lrde.epita.fr/

version in Section 5 to enhance readability and clarify the rationale behind selecting which indistinguishability relations to break.

Having discussed the theoretical foundations of our approach and introduced the resulting prototype, we now turn our attention to the experiments conducted using this prototype. These experiments include applications to the remote inspection case study presented in this paper, as well as stress test scenarios designed to explore the prototype's performance and its interaction with various metrics.

6.1 Experimental results

We evaluated our implementation from three distinct perspectives.

First, we assessed our tool based on two critical aspects: generation time and verification time. The generation time refers to the execution time required to synthesise a monitor given an LTL formula (as defined in Definition 4.8). On the other hand, the verification time concerns the execution time needed to analyse a trace of events using the synthesised monitor. It is important to distinguish between these two evaluations, as monitor generation typically occurs offline, prior to system execution. Therefore, the most crucial factor in assessing runtime verification techniques is the verification time, as it directly impacts system performance by introducing runtime overhead.

Next, we validated our technique against the remote inspection case study presented in Section 3, ensuring that our expected outcomes aligned with the event traces analysed throughout the study.

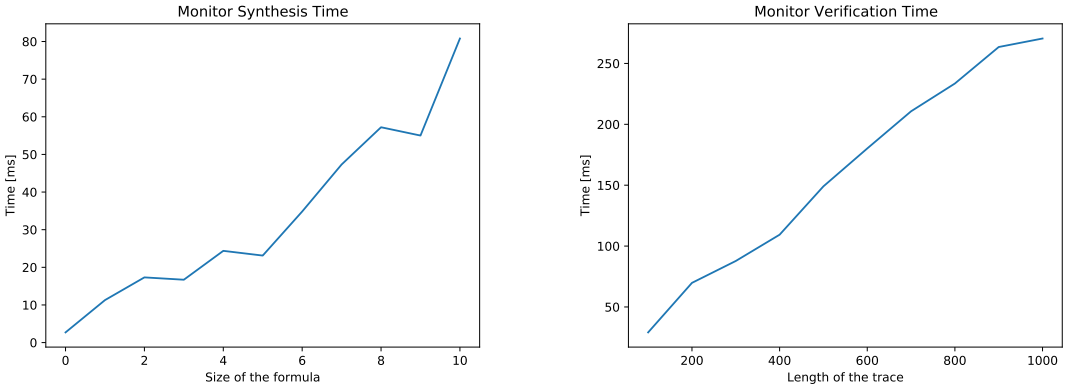
Finally, we tested our technique by varying the metrics used to assign payoffs to the indistinguishability relations. These experiments served two purposes: to demonstrate the impact of selecting effective metrics versus poor ones, and to stress-test our implementation using a set of randomly generated LTL properties and traces.

6.1.1 Overhead experiments. We began with the overhead experiments, conducting tests for both key aspects: monitor synthesis and verification. For the synthesis experiments, we varied the size of the LTL formula, defined by the number of operators within it. The size was chosen as the target for these experiments because it directly drives the monitor's generation[9].

For the verification experiments, we varied the length of the trace of events to be analysed. The trace length was selected because it is the only factor that influences the monitor's verification time. This can be easily understood by considering that once the FSM is generated, it remains unchanged. Its size is fixed, determined by the formula's complexity. Therefore, at runtime, the only variable affecting verification time is the length of the trace, which consists of events generated during system execution.

Figure 6 presents the results of our experiments, where both LTL formulas and traces were randomly generated. Specifically, Figure 6a shows the execution time required to synthesise a monitor given an LTL formula, while Figure 6b displays the execution time needed to analyse a trace of events using the synthesised monitor. In Figure 6a, the x-axis represents the size of the LTL formula, and the y-axis shows the execution time in milliseconds. As anticipated, the execution time for monitor synthesis increases exponentially with the size of the formula. In Figure 6b, the x-axis corresponds to the length of the event trace, and the y-axis again represents execution time in milliseconds. Notably, the execution time grows linearly with the trace length, which is critical for using the monitor at runtime while the system is operational. Since the execution time is linear relative to the trace length, the time required for the monitor to

[9]It is important to note that steps (iii) and (vi) in Figure 3 are particularly computationally expensive, requiring exponential time relative to the formula size.



(a) Time to synthesise a monitor.

(b) Time to verify a trace.

Fig. 6. Experimental results.

analyse each individual event in the trace remains constant. This ensures that the monitor can incrementally analyse events as they are generated by the system during runtime[10].

Remark 6.2. *It is important to note that the synthesis and execution process is identical for all imperfect information monitors, including the extended LTL monitor, the active monitor, and the reactive monitor. In essence, the active and reactive monitors are equivalent to the imperfect information monitor (as defined in Definition 4.8). In fact, they merely permit the breaking of indistinguishability relations. However, this modification does not impact the performance, synthesis, and execution of the underlying monitor.*

6.1.2 Experiments on the case study. The second set of experiments we conducted focused on the case study and its formal properties. Specifically, we tested our implementation against each property presented in Section 3 using all the types of monitors discussed in this paper. The results of these experiments are summarised in Table 1. For each property, we considered the global trace of events described in Example 4.6. Depending on the monitor type used, we reported the corresponding visible trace as observed by that monitor.

For instance, the standard monitor would observe the imperfect information trace without recognising that the missing events are not false but simply absent. In contrast, imperfect information monitors would consider both explicit events (as defined in Definition 4.4) and the equivalence classes resulting from the indistinguishability relation. Additionally, for active and reactive monitors, we reported the visible trace that results from breaking indistinguishability relations according to the metric outlined in Example 5.1.

Finally, in Table 1, we documented the outcomes produced by each monitor for each property, considering the given trace, thus validating the results discussed throughout the paper. Notably, we observed that the use of imperfect information monitors corrected the erroneous outcomes for properties φ_3 and ψ_3, which the standard monitor incorrectly classified as $\top$ and $\bot$, respectively.

Moreover, Table 1 illustrates the effect of using a reactive monitor in the verification of $\psi = \psi_1 \vee \psi_2$. The imperfect information monitor could at best return $?_{\mathcal{T}}$, similar to the active monitor. The active monitor did not improve the

[10]By "incrementally", we mean that the monitor analyses events one by one, as opposed to offline runtime verification where the monitor expects the entire trace at once.

Global trace	Visible trace	φ_1	φ_2	φ_3	ψ_1	ψ_2	ψ_3	$\psi_1 \vee \psi_2$
LTM								
$\{\}$	$\{\}$							
$\{\gamma, b^1, c\}$	$\{b^1\}$							
$\{\gamma, c, mb, b^2\}$	$\{mb, b^2\}$	?	?	$\top$	?	?	$\bot$	?
$\{c\}$	$\{\}$							
$\{w\}$	$\{w\}$							
LTL monitors with imperfect information								
$\{\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, b^1, c\}$	$\{b^1_\top, b^2_\bot, b^3_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, c, mb, b^2\}$	$\{b^1_\bot, b^2_\top, b^3_\bot, mb_\top, w_\bot\}$	$?_\not\bot$	$?_\not\bot$	$?_\not\bot$	$?_\not\top$	$?_\not\top$	$?_\not\top$	$?_\not\top$
$\{c\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$							
$\{w\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\top\}$							
Active monitor 1								
$\{\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, b^1, c\}$	$\{b^1_\top, b^2_\bot, b^3_\bot, c_\top, s_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, c, mb, b^2\}$	$\{b^1_\bot, b^2_\top, b^3_\bot, c_\top, s_\bot, mb_\top, w_\bot\}$	$\top$	$?_\not\bot$	?	$\bot$	$?_\not\top$	$?_\not\top$	$?_\not\top$
$\{c\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, c_\top, s_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$							
$\{w\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, c_\bot, s_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\top\}$							
Active monitor 2								
$\{\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, b^1, c\}$	$\{b^1_\top, b^2_\bot, b^3_\bot, \alpha_\bot, \beta_\bot, \gamma_\top, mb_\bot, w_\bot\}$							
$\{\gamma, c, mb, b^2\}$	$\{b^1_\bot, b^2_\top, b^3_\bot, \alpha_\bot, \beta_\bot, \gamma_\top, mb_\top, w_\bot\}$	$\top$	$?_\not\bot$	?	$?_\not\top$	$\bot$	$?_\not\top$	$?_\not\top$
$\{c\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\bot\}$							
$\{w\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\top\}$							
Reactive monitor								
$\{\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, \alpha_\bot, \beta_\bot, \gamma_\bot, mb_\bot, w_\bot\}$							
$\{\gamma, b^1, c\}$	$\{b^1_\top, b^2_\bot, b^3_\bot, \alpha_\bot, \beta_\bot, \gamma_\top, mb_\bot, w_\bot\}$							
$\{\gamma, c, mb, b^2\}$	$\{b^1_\bot, b^2_\top, b^3_\bot, c_\top, s_\bot, mb_\top, w_\bot\}$	$\top$	$?_\not\bot$	?	$\bot$	$\bot$	$?_\not\top$	$\bot$
$\{c\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, c_\top, s_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\bot\}$							
$\{w\}$	$\{b^1_\bot, b^2_\bot, b^3_\bot, [\gamma_{cs}]_\bot, [\gamma_{\alpha\beta\gamma}]_\bot, mb_\bot, w_\top\}$							

Table 1. Results on applying the prototype implementation on the case study. Note that, Active monitor 1 breaks γ_{cs} while Active monitor 2 breaks $\gamma_{\alpha\beta\gamma}$.

outcome for ψ because the bound **b** was set to 3, forcing the monitor to choose between breaking one equivalence class (e.g., γ_{cs}) or the other (e.g., $\gamma_{\alpha\beta\gamma}$), but not both. As a result, the reactive monitor achieved better outcomes than its counterparts by dynamically adapting to and reacting to new events. By doing so, and with a time window set to 2 (as demonstrated in Example 5.4), the reactive monitor was able to first break one equivalence class ($\gamma_{\alpha\beta\gamma}$) and then the other ($\gamma_{cs}$) in two separate steps during the trace of events analysed.

6.1.3 Experiments on metrics. The final set of experiments focused on the impact of different metrics on the results obtained by the monitors. Since these experiments were solely concerned with the influence of metrics, we opted to use active monitors instead of reactive ones. This choice was made because the objective was to isolate the effect of metrics on the monitoring process, whereas the impact of reactive behaviour was already addressed in the previous set of experiments.

Specifically, we tested four different metrics, as detailed in Table 2 and Table 3. We generated 10,000 random LTL formulas and verified them against 100 randomly generated traces of events, resulting in 1,000,000 executions for each metric analysed. It is important to note that while the formulas and traces were randomly generated, they were kept consistent across the different experiments. This ensured that each metric was tested on the same set of LTL formulas and traces, allowing for a fair comparison.

Metric	**Total**	$\top$	$\bot$	uu	?	$?_{\not\bot}$	$?_{\not\top}$
$metric_0$	1000000	175793	275205	29676	463563	29873	25890
$metric_1$	1000000	165239	280671	16983	481076	26978	29053
$metric_2$	1000000	173899	278783	**16875**	480616	29066	20761
$metric_3$	1000000	163404	284628	17498	478891	24701	30878

Table 2. Comparison of Metrics.

Table 2 presents the results of these experiments, showing the number of times each outcome was returned by the monitor for each metric. Table 3 provides the same results in percentage form, offering a clearer understanding of the metrics' influence.

Metric	**Total**	$\top$	$\bot$	uu	?	$?_{\not\bot}$	$?_{\not\top}$
$metric_0$	1000000	17.58%	27.52%	2.97%	46.36%	2.99%	2.59%
$metric_1$	1000000	16.52%	28.07%	1.70%	48.11%	2.70%	2.91%
$metric_2$	1000000	17.39%	27.88%	**1.69%**	48.06%	2.91%	2.08%
$metric_3$	1000000	16.34%	28.46%	1.75%	47.89%	2.47%	3.09%

Table 3. Comparison of Metrics in Percentages.

We labelled the metrics from 0 to 3, referring to each as $metric_i$ for brevity. Among these, $metric_2$ corresponds to the metric introduced in Example 5.1, while the others are variations that assign different weights to temporal operators (see the Appendix for the detailed description). Specifically, $metric_0$ serves as a baseline metric, as it assigns payoffs to atomic propositions without giving particular importance to any of them. We included this baseline metric to demonstrate how selecting a more carefully considered metric can significantly improve the monitoring process. This improvement is evident in Table 2, where $metric_0$ shows the poorest performance, with the highest number of undefined outcomes. In contrast, $metric_2$ performs the best, yielding the fewest undefined outcomes. Notice that, the undefined value is the worst outcome between the six truth values. In fact, it gives a definitive result without providing any information on the analysed formula.

Remark 6.3. *It is important to note that while $metric_2$ performs better than $metric_0$ in terms of reducing the number of undefined outcomes, the percentage of such outcomes remains minimal compared to the other monitor results (with ? being the most frequently returned outcome). This indicates that although selecting a well-suited metric can indeed reduce the proportion of undefined outcomes, its impact is more pronounced in scenarios where the metric is contextually relevant. In the experiments summarised in Table 2 and Table 3, the random generation of LTL formulas and traces demonstrates that the choice of a metric does not significantly affect the overall distribution of outcomes. However, it does play a crucial role in minimising the occurrence of the least desirable outcome, namely the undefined result.*

7 Related work

Several works address runtime verification in scenarios involving uncertainty; for a comprehensive overview, we refer to a recent survey on the topic [33].

The work most closely related to ours is [35], where Past-Time LTL is verified at runtime under uncertainty regarding the observed events. In that study, verification is performed on abstract traces of events, where not all concrete events are present—only samples taken at specific time intervals. The uncertainty arises from unknown event interleaving,

whereas in our approach, it stems from indistinguishability relations between events. Unlike [35], we do not sample events; instead, the uncertainty is determined by the monitor's visibility. Consequently, our abstraction focuses not on the order of events in a trace but on the types of events the trace contains.

Another closely related work to ours is presented in [24], where the authors propose an approach to RV in scenarios involving transient event loss—where events are temporarily lost, their quantity is known, but their content remains unknown. The objective of their work is to demonstrate that certain properties can still be monitored despite these losses, provided that subsequent valid events are observed. They denote lost events using the symbol χ and express the properties using LTL. These properties are then translated into an RV-LTL monitor, which is represented as a finite-state automaton. To create a loss-tolerant monitor, they introduce a new state that manages the lossy elements. The monitor outputs a verdict of ? (analogous to our *uu* verdict) upon encountering a lost event but continues its operation based on the subsequent valid events. Unlike our approach, which also considers online monitoring and the management of imperfect information through active and reactive monitors, the approach in [24] is confined to an offline setting and does not address the handling of imperfect information.

The works in [6–8] and ours both address RV under challenging conditions, yet they differ significantly in their approaches to handling imperfect information and the types of monitors employed. The methods in [6–8] emphasise runtime verification techniques that manage incomplete or conflicting logs, imprecise timestamps, and out-of-order data streams, ensuring effective monitoring despite these adversities. In contrast, our work advances standard RV by explicitly addressing imperfect information through the use of indistinguishability relations, and by developing a novel "rational monitor" capable of dynamically adjusting its behaviour based on the information it receives. This proactive approach contrasts with the more static methods used in [6–8]. Furthermore, while the methods in [6–8] are typically applied within traditional RV frameworks, often in compliance monitoring or real-time systems, our approach is specifically designed for autonomous systems. This emphasis on a monitor that remains effective even under severely limited or faulty information makes our approach particularly suitable for complex, real-world environments such as autonomous robotics.

The work in [1] addresses the monitorability of branching-time logics that incorporate silent actions, focusing on how these actions influence the ability to verify certain properties at runtime. Their research is particularly concerned with the challenges posed by silent actions in RV and develops a framework to assess the monitorability of specifications in the presence of these actions. In contrast, our work expands on the concept of RV by addressing scenarios where the information available is imperfect, utilising indistinguishability relations to manage and verify systems under these conditions. Furthermore, we introduce the consideration of rational aspects in RV, particularly under conditions of imperfect information. While both studies contribute to the advancement of RV techniques, our approach offers a broader perspective by encompassing various types of imperfect information, whereas [1] is more specialised, dealing with the specific implications of silent actions within a RV context.

In a different line of research, works such as [4, 5, 25, 27, 32] address uncertainty in verification caused by the absence of information. In these studies, event traces may contain gaps, meaning that the monitor cannot observe system behaviour at certain points during execution. This issue has been addressed in various ways, typically by filling these gaps with potential events. Given the uncertainty about which events actually occurred during these gaps, these approaches often rely on probabilistic methods to infer the missing events. These works differ fundamentally from ours because we do not assume any missing information—there are no gaps in our traces. Instead, our uncertainty arises from the monitor's inability to distinguish between certain events due to indistinguishability relations.

A more recent work on RV with uncertainty is [34], where uncertainty is abstracted using multi-traces instead of standard uni-traces. A multi-trace allows multiple evaluations for the same atomic proposition within the trace. The authors propose a monitor designed to handle such multi-traces and prove its soundness. Similar to [4, 5, 25, 27, 32], this work also focuses on missing events, although it considers partially missing events as well.

Unlike our approach, all the aforementioned works explicitly represent the notion of uncertainty (e.g., through gaps). When traces contain concrete events, these works typically adhere to standard semantics. Our approach, by contrast, is less intrusive, building on the existing RV pipeline for verifying LTL properties without introducing gaps. We focus on extending the standard RV technique for LTL to handle scenarios where the monitor has imperfect information about the system. From an engineering standpoint, our method enhances the standard LTL approach for use in cases of imperfect information, while the other works in the literature primarily propose entirely new techniques to manage the absence of information, often caused by noise or technical issues.

8 Conclusions and future work

In this paper, we introduced several extensions to the standard LTL runtime verification approach. We addressed the challenge of imperfect information at the monitor level and demonstrated how this lack of information can cause a standard LTL monitor to produce incorrect verdicts. To mitigate this issue, we proposed an extended version of the LTL monitoring process specifically designed to handle imperfect information. However, monitors with imperfect information yield fewer verdicts than those with perfect information. To address this limitation, we introduced the novel concept of rational monitors. In particular, we defined two classes of rational monitors: active and reactive, which are designed to improve the handling of imperfect information at the monitoring level. We provided a theoretical framework for understanding imperfect information through the use of equivalence classes, alongside the concepts of active and reactive monitors, and examined how imperfect information affects the verification of LTL properties. Additionally, we presented a Python prototype implementing our approach and demonstrated its application on a relevant case study, alongside additional experiments designed to stress-test the implementation.

For future work, we envision three main directions. First, we plan to further investigate the role of metrics in the imperfect information runtime verification process. Our goal is to identify a metric that offers the best possible trade-off for verifying LTL formulas that minimises the percentage of undefined outcomes while preserving other definitive truth outcomes. Second, we aim to extend the concept of rational monitors to a multi-agent setting. This extension could have significant implications for real-world applications, particularly in Internet of Things (IoT) systems, where individual components are often monitored independently. Due to the distributed nature of IoT systems, these components frequently lack complete knowledge of the entire system. In such scenarios, a variant of our approach could be highly effective. Third, although this paper primarily focuses on the foundational aspects of our approach, we have also presented a prototype as a proof of concept. Naturally, to further develop and enhance the practical applicability of our approach – while preserving the same engineering principles and methodology – we can optimize it to improve performance. In particular, since monitor synthesis is often the bottleneck in runtime verification approaches, efficiency could be improved by building on the contributions of [16]. Finally, given the relevance of our approach within the field of distributed artificial intelligence – particularly in Multi-Agent Systems – we plan to integrate it into the VITAMIN verification framework [19–21].

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. 2017. Monitoring for Silent Actions. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India (LIPIcs, Vol. 93)*, Satya V. Lokam and R. Ramanujam (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:14. https://doi.org/10.4230/LIPICS.FSTTCS.2017.7

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

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

[4] Ezio Bartocci and Radu Grosu. 2013. Monitoring with uncertainty. In *Proceedings Third International Workshop on Hybrid Autonomous Systems, HAS 2013, Rome, Italy, 17th March 2013 (EPTCS, Vol. 124)*, Luca Bortolussi, Manuela-Luminita Bujorianu, and Giordano Pola (Eds.). 1–4. https://doi.org/10.4204/EPTCS.124.1

[5] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. 2012. Adaptive Runtime Verification. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7687)*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer, 168–182. https://doi.org/10.1007/978-3-642-35632-2_18

[6] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. 2012. Monitoring Compliance Policies over Incomplete and Disagreeing Logs. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7687)*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer, 151–167. https://doi.org/10.1007/978-3-642-35632-2_17

[7] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. 2014. On Real-Time Monitoring with Imprecise Timestamps. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8734)*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer, 193–198. https://doi.org/10.1007/978-3-319-11164-3_16

[8] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2017. Runtime Verification of Temporal Properties over Out-of-Order Data Streams. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 356–376. https://doi.org/10.1007/978-3-319-63387-9_18

[9] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2006. Monitoring of Real-Time Properties. In *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, Vol. 4337)*, S. Arun-Kumar and Naveen Garg (Eds.). Springer, 260–272. https://doi.org/10.1007/11944836_25

[10] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 14 (Sept. 2011), 64 pages. https://doi.org/10.1145/2000799.2000800

[11] Francesco Belardinelli, Angelo Ferrando, and Vadim Malvone. 2023. An abstraction-refinement framework for verifying strategic properties in multi-agent systems with imperfect information. *Artif. Intell.* 316 (2023), 103847. https://doi.org/10.1016/J.ARTINT.2022.103847

[12] Francesco Belardinelli, Alessio Lomuscio, Vadim Malvone, and Emily Yu. 2022. Approximating Perfect Recall when Model Checking Strategic Abilities: Theory and Applications. *J. Artif. Intell. Res.* 73 (2022), 897–932. https://doi.org/10.1613/jair.1.12539

[13] Edmund M Clarke. 1997. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 54–56.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[15] Alexandre Duret-Lutz and Denis Poitrenaud. 2004. SPOT: An Extensible Model Checking Library Using Transition-Based Generalized Büchi Automata. In *12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 4-8 October 2004, Vollendam, The Netherlands*, Doug DeGroot, Peter G. Harrison, Harry A. G. Wijshoff, and Zary Segall (Eds.). IEEE Computer Society, 76–83. https://doi.org/10.1109/MASCOT.2004.1348184

[16] Cindy Eisner and Dana Fisman. 2018. *Functional Specification of Hardware via Temporal Logic*. Springer International Publishing, Cham, 795–829. https://doi.org/10.1007/978-3-319-10575-8_24

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

[18] Angelo Ferrando and Vadim Malvone. 2022. Runtime Verification with Imperfect Information Through Indistinguishability Relations. In *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13550)*, Bernd-Holger Schlingloff and Ming Chai (Eds.). Springer, 335–351. https://doi.org/10.1007/978-3-031-17108-6_21

[19] Angelo Ferrando and Vadim Malvone. 2024. Hands-on VITAMIN: A Compositional Tool for Model Checking of Multi-Agent Systems. In *Proceedings of the 25th Workshop "From Objects to Agents", Bard (Aosta), Italy, July 8-10, 2024 (CEUR Workshop Proceedings, Vol. 3735)*, Marco Alderighi, Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi (Eds.). CEUR-WS.org, 148–160. https://ceur-ws.org/Vol-3735/paper_12.pdf

[20] Angelo Ferrando and Vadim Malvone. 2025. VITAMIN: A Compositional Framework for Model Checking of Multi-Agent Systems. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025 - Volume 1, Porto, Portugal, February 23-25, 2025*, Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik (Eds.). SCITEPRESS, 648–655. https://doi.org/10.5220/0013349600003890

[21] Angelo Ferrando and Vadim Malvone. 2025. VITAMIN: VerIficaTion of A MultI ageNt system. In *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025)*. To appear.

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

[23] JE Hopcroft. 2001. Introduction to Automata Theory, Languages, and Computation.

[24] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. 2017. Runtime verification of LTL on lossy traces. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1379–1386. https://doi.org/10.1145/3019612.3019827

[25] Kenan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott D. Stoller, and Radu Grosu. 2013. Runtime Verification with Particle Filtering. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8174)*, Axel Legay and Saddek Bensalem (Eds.). Springer, 149–166. https://doi.org/10.1007/978-3-642-40787-1_9

[26] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems*. Springer. https://doi.org/10.1007/978-3-540-24777-7

[27] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2019. Runtime Verification for Timed Event Streams with Partial Information. In *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11757)*, Bernd Finkbeiner and Leonardo Mariani (Eds.). Springer, 273–291. https://doi.org/10.1007/978-3-030-32079-9_16

[28] Martin Leucker and Christian Schallhart. 2009. 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

[29] Jose P. Miguel, David Mauricio, and Glen Rodriguez. 2014. A Review of Software Quality Models for the Evaluation of Software Products. *CoRR* abs/1412.2977 (2014). arXiv:1412.2977 http://arxiv.org/abs/1412.2977

[30] 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, 46–57. https://doi.org/10.1109/SFCS.1977.32

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

[32] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. 2011. Runtime Verification with State Estimation. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7186)*, Sarfraz Khurshid and Koushik Sen (Eds.). Springer, 193–207. https://doi.org/10.1007/978-3-642-29860-8_15

[33] Rania Taleb, Sylvain Hallé, and Raphaël Khoury. 2023. Uncertainty in runtime verification: A survey. *Comput. Sci. Rev.* 50 (2023), 100594. https://doi.org/10.1016/J.COSREV.2023.100594

[34] Rania Taleb, Raphaël Khoury, and Sylvain Hallé. 2021. Runtime Verification Under Access Restrictions. In *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*, Simon Bliudze, Stefania Gnesi, Nico Plat, and Laura Semini (Eds.). IEEE, 31–41. https://doi.org/10.1109/FormaliSE52586.2021.00010

[35] Shaohui Wang, Anaheed Ayoub, Oleg Sokolsky, and Insup Lee. 2011. Runtime Verification of Traces under Recording Uncertainty. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7186)*, Sarfraz Khurshid and Koushik Sen (Eds.). Springer, 442–456. https://doi.org/10.1007/978-3-642-29860-8_35

[36] M. Wooldridge and A. Rao (Eds.). 1999. *Foundations of Rational Agency*. Kluwer Academic Publishers.

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

Appendix

We report the $metric_0$ as used in Section 6

$$
\begin{aligned}
metric_q(p) &= 0 \; with \; p \neq q \\
metric_p(p) &= 1 \\
metric_{\neg p}(p) &= 1 \\
metric_{\varphi \vee \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\varphi \wedge \varphi'}(p) &= min(metric_{\varphi}(p), metric_{\varphi'}(p)) \\
metric_{\varphi \Rightarrow \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\bigcirc \varphi}(p) &= 0.1 \times metric_{\varphi}(p) \\
metric_{\varphi \, \mathrm{U} \, \varphi'}(p) &= 0.9 \times metric_{\varphi}(p) + 0.1 \times metric_{\varphi'}(p) \\
metric_{\varphi \, \mathrm{R} \, \varphi'}(p) &= 0.9 \times metric_{\varphi}(p) + 0.1 \times metric_{\varphi'}(p)
\end{aligned}
$$

We report the $metric_1$ as used in Section 6

$$
\begin{aligned}
metric_q(p) &= 0 \; with \; p \neq q \\
metric_p(p) &= 1 \\
metric_{\neg p}(p) &= 1 \\
metric_{\varphi \vee \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\varphi \wedge \varphi'}(p) &= max(metric_{\varphi}(p), metric_{\varphi'}(p)) \\
metric_{\varphi \Rightarrow \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\bigcirc \varphi}(p) &= 0.5 \times metric_{\varphi}(p) \\
metric_{\varphi \, \mathrm{U} \, \varphi'}(p) &= 0.3 \times metric_{\varphi}(p) + 0.7 \times metric_{\varphi'}(p) \\
metric_{\varphi \, \mathrm{R} \, \varphi'}(p) &= 0.5 \times metric_{\varphi}(p) + 0.5 \times metric_{\varphi'}(p)
\end{aligned}
$$

We report the $metric_3$ as used in Section 6

$$
\begin{aligned}
metric_q(p) &= 0 \; with \; p \neq q \\
metric_p(p) &= 1 \\
metric_{\neg p}(p) &= 1 \\
metric_{\varphi \vee \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\varphi \wedge \varphi'}(p) &= max(metric_{\varphi}(p), metric_{\varphi'}(p)) \\
metric_{\varphi \Rightarrow \varphi'}(p) &= (metric_{\varphi}(p) + metric_{\varphi'}(p))/2 \\
metric_{\bigcirc \varphi}(p) &= 1.0 \times metric_{\varphi}(p) \\
metric_{\varphi \, \mathrm{U} \, \varphi'}(p) &= 0.3 \times metric_{\varphi}(p) + 0.7 \times metric_{\varphi'}(p) \\
metric_{\varphi \, \mathrm{R} \, \varphi'}(p) &= 0.3 \times metric_{\varphi}(p) + 0.7 \times metric_{\varphi'}(p)
\end{aligned}
$$

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009