A Hierarchy of Monitoring Properties for Autonomous Systems

Sebastian Schirmer and Christoph Torens and Johann C. Dauer
DLR German Aerospace Center, Institute of Flight Systems, Dept. Unmanned Aircraft, Braunschweig, Germany

Jan Baumeister and Bernd Finkbeiner
CISPA Helmholtz Center for Information Security, Reactive Systems Group, Saarbrücken, Germany

Kristin Y. Rozier
Iowa State University, Laboratory for Temporal Logic, Iowa, USA

Monitoring capabilities play a central role in mitigating safety risks of current, and especially future autonomous aircraft systems. These future systems are likely to include complex components such as neural networks for environment perception, which pose a challenge for current verification approaches; they are considered as black-box components. To assure that these black-boxes comply with their specification, they must be monitored to detect violations during execution with respect to their input and output behaviors. Such behavioral properties often include more complex aspects such as temporal or spatial notions. The outputs can also be compared to data from other assured sensors or components of the aircraft, making monitoring an integral part of the system, which ideally has access to all available resources to assess the overall health of the operation. Current approaches using handwritten code for monitoring functions run the risk of not being able to keep up with these challenges. Therefore, in this paper, we present a hierarchy of monitoring properties that provides a perspective for overall health. We also present a categorization of monitoring properties and show how different monitoring specification languages can be used for formalization. These monitoring languages represent a higher abstraction of general-purpose code and are therefore more compact and easier for a user to write and read, and we can validate their implementations independently from the systems they reason about. They improve the maintainability of monitoring properties that is required to handle the increased complexity of future autonomous aircraft systems.

I. Introduction

Future autonomous aircraft systems promise to support a multitude of applications such as cargo transportation, inspection flights, and urban air mobility. These business cases depend on a high degree of automation to work. Interestingly, however, the performance of the automation functions is no longer the main concern, since well-performing vision-based autonomous flying systems already exist; instead the challenge lies in ensuring the safety of these functions during operation [1]. To safely bound the behavior of such a complex automation function, ASTM International proposes a run-time assurance architecture [2], simplified in Figure 1. At its core lies a safety monitor that switches to an assured recovery control function when it detects a violation of a safety property. Here, ASTM defines *assured* as an "attribute of an entity for which sufficient evidence exists to demonstrate that an acceptable level of rigor has been met" [2]. These safety properties can range from simple threshold checks to more complex properties that include temporal and spatial aspects. Further, meaningful safety properties often compare outputs of a monitored function to other assured sensors or components of the aircraft, making monitoring an integral part of the system that requires a holistic system perspective.

Since current approaches using handwritten code for monitoring functions involve writing specialized code each time that can be tricky to validate, and run the risk of not being able to keep up with increased system complexity, we promote the use of standardized, formal monitoring specification languages in this paper. Specification languages directly support notions like time and, therefore, allow system designers to concisely formalize more complex monitoring properties like temporal properties. They are easier to read and write than lower-level code, present opportunities for independent validation and re-use, and can serve as inputs to verified engines that generate the lower-level code automatically. Hence formal specifications tend to be less error-prone and easier to maintain. Further, based on a formalized specification that consists of properties to be monitored, several existing tools can automatically generate monitor implementations that come with additional proven guarantees, such as bounds on the memory consumption. In order to motivate the use of such a monitoring specification language, we first report on work that successfully applied monitoring based on specification languages. Next, we present a hierarchy of monitoring properties that ranges from low-level sensor properties to high-level operation properties, giving a holistic perspective on possibilities for system monitoring. Further, we categorize different types of monitoring properties and showcase different monitoring specification languages. Finally, we discuss the advantages and disadvantage of using such formal languages.

Related Work: Aerospace Regulations

Aviation standards and regulations offer a perspective on target systems that informs their monitoring. For instance, Aerospace Recommended Practice (ARP) 4754A [3] from SAE International is the guidelines document for development of civil aircraft and systems. Additionally, SAE International's ARP4761 [4] provides guidelines for conducting the safety assessment process on civil airborne systems and equipment. The terms used in both documents for developing an aircraft as well as performing a safety assessment on the different process levels are item, system, and aircraft. An *item*, is one or more hardware and/or software elements treated as a unit. An item is the lowest level of abstraction during development. A *system* is a combination of inter-related items arranged to perform a specific function. A system therefore represents a higher abstraction level than an item. The *aircraft* is the largest entity and therefore the highest abstraction level in the development process of traditional aviation. Another important aspect in this context, which is required between these process levels, is establishing bidirectional traceability. Requirements must be traced to functions and systems as well as verification steps. As a result, it makes sense to organize runtime monitoring in a similar hierarchy, considering these development process levels and respecting the required traceability.

Another regulatory document, the Jarus Specific Operation Safety Assessment (SORA) [5] provides guidelines for conducting safety assessments of the recently-introduced specific categories of aircraft defined by the European Union Aviation Safety Agency (EASA). The relevant term introduced in SORA is *operation*. The idea is that the operation can be safe, even if the aircraft is not. For example, we consider a drone that is capable of photo and video documentation. The drone is robust, but there is a possibility that every thousand hours the system will fail and the drone will fall to the ground. From a certification perspective, the aircraft is not safe. However, the operation may be safe if flights only occur in unpopulated areas, so the *operation* is therefore at an even higher level of abstraction than the *aircraft*.

The above-mentioned standards do not specifically address autonomy, such as adaptations to regulations for certification of aerospace systems incorporating different levels of autonomy [6]. However, in the context of autonomy, artificial intelligence, and machine learning in particular, EASA has recently published new guidance documents [7–10]. This is the first guidance document for the certification of machine learning (ML). There are four building blocks for assuring the safety of ML components. Via safety assessments, learning assurance, and AI explainability, the safety of the ML components can achieve certification. However, it is not completely possible to assure safety. As a result, monitoring is necessary for further safety assurance. We can meet several objectives inside this guidance document by monitoring aspects of input data. For example, one of these building blocks is called AI Safety Risk Mitigation. Basically, this requires monitoring the input and output of the ML component for unsafe actions in order to mitigate risk, similar to recovery control function in ASTM F3269-21. Further details on the analysis of monitoring aspects in the standard appear in [11].

ASTM F3269-21 not only describes a reference architecture for assuring the safety of black-box systems, or systems

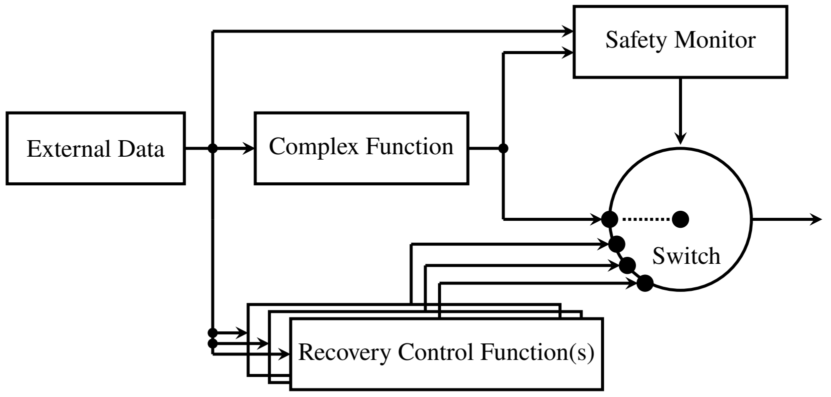


Fig. 1 A simplified version of the run-time assurance architecture proposed by ASTM F3269-21 to safely bound a complex function using a safety monitor.

that are too complex to comply to traditional verification standards, but also introduces aspects of monitor coverage, recovery function coverage, and run-time assurance system coverage. These aspects are required to guarantee that the implemented architecture covers all critical tasks.

All of these standards provide guidance on what a monitor must achieve, but none provides guidance on how a monitor should be implemented. Therefore, we discuss the advantages and disadvantages of formal specification languages and how an implementation based on formal specifications addresses several of the mentioned concerns. The research field of Runtime Monitoring is also referred to as *Runtime Verification* (RV). For more information on this field, see [12], which classifies RV tools within a high-level taxonomy of concepts. Comparatively, this paper provides a more concrete, domain-specific perspective on specification languages for runtime monitoring in aviation.

II. Runtime Monitoring Applied to Larger Cyber-physical Systems

We highlight three works that have successfully applied monitoring specification languages for developing and assuring the safety of larger cyber-physical systems.

R2U2 Framework

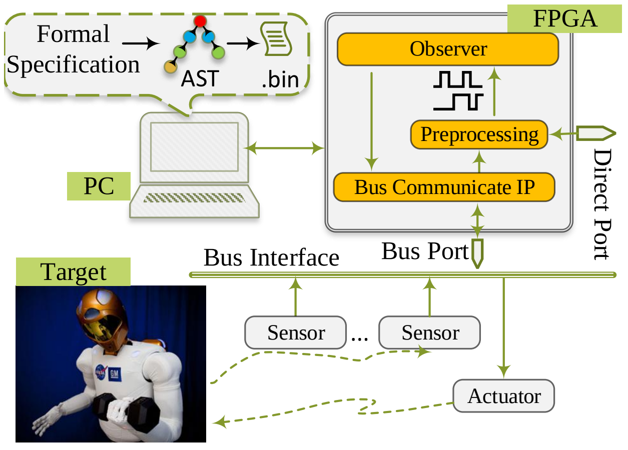
R2U2[*] was first designed for the NASA SwiftUAS which is a 13-foot-wingspan, all-electric experimental platform based upon a high-performance sailplane to monitor requirements derived from National Aeronautics and Space Administration (NASA) and Federal Aviation Administration (FAA) processes and standards [13]. R2U2 is named after the requirements it is designed to uphold [14] — a Responsive, Realizable, and Unobtrusive Unit dedicated to monitoring a safety-critical cyber-physical system (CPS). The requirements ranged from value checks, over cross checks, to flight rules.

Figure 2 gives a R2U2 tool chain overview that receives a specification of these requirements in form of a temporal logic formula and a Bayesian Network. The temporal logic formula represents the monitor module and the Bayesian Network the consecutive health reasoning module that uses monitor outputs. Both modules are synthesized on an FPGA and integrated into the CPS. As temporal logic, a variant of Linear Temporal Logic (LTL) called Mission-time LTL [15] (MLTL) is supported. LTL is propositional logic extended by temporal operators that allow to reason over sequences of system events. For instance, the LTL formula $\texttt{cmd} \wedge \textbf{next}\ \texttt{cmd}$ represents that a `cmd` event is expected in the current step and another `cmd` event is expected in the next step, the LTL formula $\neg\texttt{dangerous\_cmd}\ \textbf{until}\ \texttt{alt} \geq 200\ m$ states that no `dangerous_cmd` shall be given *until* an altitude of more than 200 meters is reached, the LTL formula $\square(\texttt{vel<10 m/s})$ states that the velocity shall *always* be smaller than 10 m/s, and $\Diamond\ \texttt{land\_cmd}$ states that *eventually* a landing command shall be given. The latter property is rather vague and can always be extended to a trace that satisfies the property. This can be bound using Mission-time LTL which adds time bounds $J = [t, t']$ to the LTL operators where $t, t' \in \mathbb{N}_0$. For instance, $\Diamond_{[0,30]}\ \texttt{land\_cmd}$ states that *eventually* within 30 time units the landing command must eventually be given to satisfy the property. Using these operators, the requirements derived from NASA and FAA documents could be compactly and rigorously expressed. For instance, the flight rule "after receiving a command (`cmd`) for takeoff, the SwiftUAS must reach an altitude of 600 ft within 40 s" is stated as:
$\square((\texttt{cmd = takeoff}) \rightarrow \Diamond_{[0,40]}(\texttt{alt} \geq 600))$.

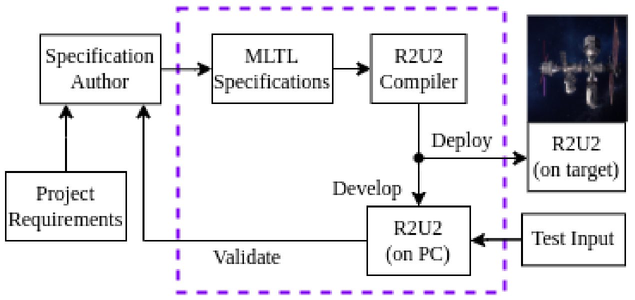
Safe Operation Monitoring with RTLola

In 2015, EASA introduced three categories of operations for unmanned aircraft systems [18] -– *open*, *specific*, and *certified*. As novelty, the specific category uses a scaling of the certification rigor depending on the overall operational risks. To assess this risk the so-called *Specifc Operation Risk Assessment (SORA)* [5] is an acceptable means of compliance. The risks that determine certification requirements are subdivided into ground risk, i.e., population density in combination with the size of the aircraft, and air risk, i.e., type of airspace and traffic density. Depending on this risk, a set of operational safety objectives (OSO) vary in rigor, including airworthiness, manufacturer and operator qualification, procedures, human factors, and reliance on external services and infrastructure. Runtime monitoring plays an important role in this context, as it allows monitoring the boundary conditions of the operation to which the operational permit was granted. Intuitively, this operational monitoring can be a geofence to guarantee containment of the permitted operational volume [19][†]. Beyond that, it can contribute to many of the OSOs including the monitoring of environmental limits as well as limits of the assurance of components, e.g., data-link and GPS, environmental perception, and aircraft performance limits. In conjunction with using formal methods, the monitoring of operational conditions

[*] `http://r2u2.temporallogic.org/`
[†] https://www.dlr.de/content/de/video/2019/flugversuche-mit-dem-alaady-demonstrator-engl.html



(a) R2U2 observers, encoded (in hardware) on an FPGA, monitor internal sensor values passed over the Robonaut2 control bus [16].



(b) Design, validation, and deployment workflow of software-implemented R2U2 on board the NASA Lunar Gateway: a set of Mission-time Linear Temporal Logic (MLTL) specifications represent the Assume-Guarantee Contract project requirements; these specifications compile into an R2U2 configuration. Once the designer validates a configuration, engineers then deploy that configuration onto the target platform [17].

Fig. 2 Overview of the R2U2 tool chain [14].

directly linked to SORA was coined *safe operation monitoring* [20, 21]. More specifically, [20] investigated the use of the formal monitoring specification language RTLola instead of hand-written code. This study also showed that the focus of such a language on *what* needs to be monitored, rather than *how* any given requirement gets monitored, helped keep the specification of operation conditions concise and precise, which also simplifies future specification maintenance tasks.

RTLola is a real-time stream-based specification language[‡]. In RTLola, incoming data arrive as timed, and possibly asynchronous, data streams that stream equations translate into output streams. An equation provides two pieces of information: "when" and "with." The information "when" is given by a static filter that bounds the evaluation either to the flow of incoming data or to periodic points in time and a dynamic filter that is a Boolean stream expression that decides if the stream gets evaluated. The "with" information is a stream expression that determines the new value of the stream. As an example, consider the following stream definitions. We consider time in discretized time steps; for example, sensors produce discretized streams of input data values over a progression of time steps. The following specification gets as input the current altitude and counts the number of time steps when the altitude is above 600.

```
input alt: Float64
output counter eval @alt when alt > 600 with counter.offset(by: -1, dft: 0) + 1
```

The output stream `counter` has the static filter `@alt`, i.e., the stream is evaluated with every altitude value given to the monitor, and is therefore bounded to the information flow. Additionally, it has the dynamic filter `alt > 600`, i.e., the stream is only evaluated iff the condition is satisfied. For the evaluation, the equation accesses the previous value of the stream with the offset-operator and increments it. By design, the RTLola-language supports different operators to access values over time. These temporal accesses to stream values can be over discrete time, as seen in the example, but also over real-time like building the average over a timed window. Note that the order of stream declarations does not affect the evaluation, as it does for declarative languages. In RTLola, the order of evaluation is determined by the dependencies between streams.

Given a RTLola specification, the framework performs various static analyses to provide reliable guarantees prior to execution, e.g., to determine a memory bound or verify stream behaviors [22]. It then either directly executes the specification with the RTLola-Interpreter [23] or compiles it to software [24] or hardware [25]. The software compilation to rust provides proof-carrying code, i.e., it annotates the generated code with Viper-Annotation [24], verifying that the semantics of the code corresponds the semantics of the RTLola specification. The hardware compilation translates a RTLola specification to VHDL that is subsequently used for realization on an FPGA.

[‡]https://www.react.uni-saarland.de/tools/rtlola/

Falsification Based on a Temporal Logic Formula
Runtime monitoring specification languages are not only useful during operation but also during development of CPS. For instance, Toyota investigates the use of *Falsification* of safety properties given in temporal logic for their closed-loop system development [26, 27]. Here, the outputs of the monitor are treated as a robustness metric to guide simulation runs towards failures. The robustness value reflects the degree of satisfaction of the safety property given the current simulation run. A negative value represents a falsified property and a positive value represents a satisfied property. For instance, the relation between increasing the nominal cruise speed of a car and a safety distance property could be detected by a decreasing robustness value. This relation can then be used to falsify the safety distance property by continuously increasing the speed from one simulation run to the next. Further, a slightly modified approach can also mine parameters [28], i.e., thresholds like the maximal speed where the safety distance property is satisfied.

III. Hierarchy of Monitoring Properties for Autonomous Aircraft Systems

We establish a hierarchy of monitoring properties for autonomous aircraft systems, depicted in Figure 3. The hierarchy divides into different abstraction levels of monitoring properties that represent stages of the autonomous software stack where monitoring plays an important role in safe operation.

The proposed abstraction levels extend the abstraction levels presented in SAE ARP4761. The lowest abstraction is on the item-level, e.g., sensors that measure the physical world. Typically, monitors check outputs of sensors according to pre-defined ranges, frequencies, or signal properties. The next abstraction level is the system-level, i.e., these are properties on specific functions of the aircraft. One such function that can be monitored is the position estimation. Here, typically, one does not rely on a single sensor but on multiple, potentially dissimilar sensors; comparing values from such a set of sensors can help detect faults. The next abstraction level is the aircraft-level that assesses the current status of the aircraft. Similar to the system-level, we can cross-check redundant systems for consistency, but we can also monitor the data-flow between systems that control the aircraft. For instance, consider two systems of the aircraft: one that controls the altitude of the aircraft and another system that assesses the current position. In this case, monitoring can be used to check whether control commands are effective: “When an altitude increase is commanded, an altitude increase must be observed within three seconds.” Above the aircraft-level is the mission-level. We define as mission the objective the aircraft shall achieve, e.g., “transport goods from A to B while maintaining a distance of 200 meters to C.” Monitoring checks whether the current operational run upholds such mission properties, and, given the aircraft status gathered by the lower abstraction levels, whether the mission remains feasible. Since the objective, or the corresponding plan to achieve it, can change during flight, the next level of abstraction, the operation-level, defines rules for valid missions. One typical operation-level property is geofencing. Geofencing defines a virtual barrier beyond which aircraft are not allowed to operate. For instance, a geofence can capture the boundaries of a field within which an agricultural drone is free to operate. Violating operation-level properties often involves the activation of contingency or emergency procedures.

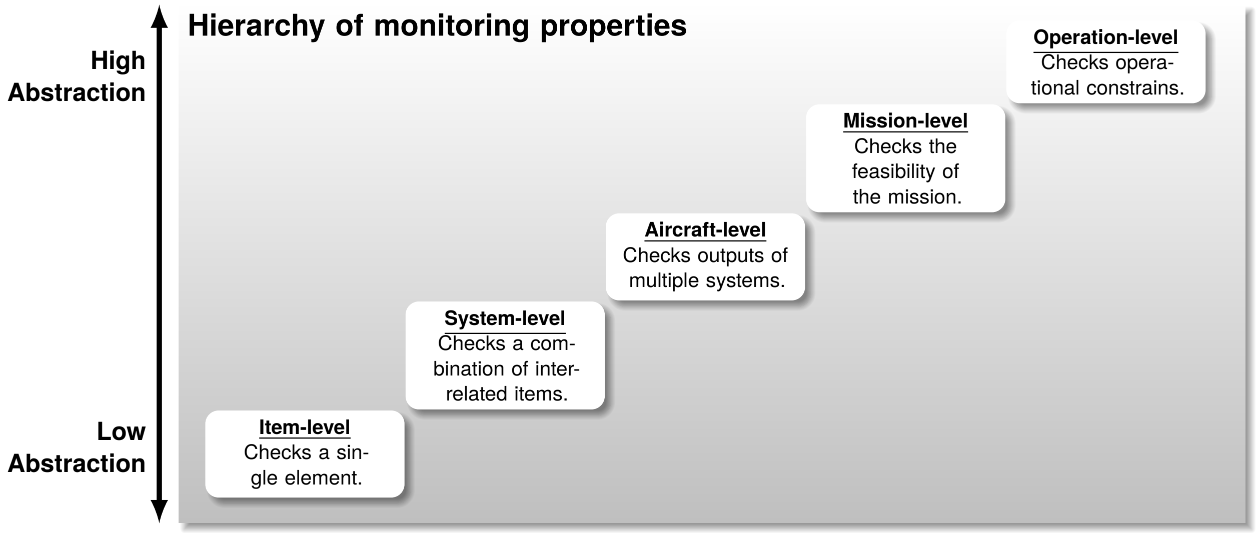

Hierarchy of monitoring properties
High Abstraction
Low Abstraction
Operation-level
Checks operational constrains.
Mission-level
Checks the feasibility of the mission.
Aircraft-level
Checks outputs of multiple systems.
System-level
Checks a combination of interrelated items.
Item-level
Checks a single element.

Fig. 3 Overview of the proposed abstraction-levels of monitoring properties for an autonomous aircraft system.

IV. Types of Monitoring Properties

Here we introduce types of monitoring properties that extend arithmetic expressions. For each type, we state example properties for the levels in the monitoring hierarchy. Further, we formalize some of the examples using different formal specification languages. We will not give a detailed description of the specification languages, but we will show their flavors and give pointers to further reading. Note that there is a trade-off between the complexity of a specification language and its expressiveness. For instance, if the specification handles asynchronous communication, then it requires additional operators to reason about asynchronously arriving data.

A. Temporal properties

Properties that reason about temporal behavior are the most common language features in RV. Section II already depicted some examples using the temporal operators $\Diamond$ and $\Box$ that represent *eventually* and *always*, respectively. Table 1 depicts some example properties for each abstraction level. Temporal properties range from properties that specify real-time properties like "Each second, the lidar sensor shall produce a value" to properties on the order of events like "The aircraft shall fullfil its mission to fly to waypoint G, reaching first waypoint A and then waypoint B in between". Latter can be formalized in LTL as $(\neg B$ **until** $A) \wedge (\neg G$ **until** $B)$ that states that B does not hold *until* A is reached and G does not hold *until* B is reached.

MLTL allows to add discrete-time bounds on the temporal operators, e.g, $\Diamond_{[0,10]}$ `landing` states that within ten time steps the `landing` must be satisfied. Both, LTL and discrete-time MLTL, assume a synchronous model for arriving data, i.e., all inputs arrive at the same time. There are also languages like RTLola that leverage this aspect to asynchronously arriving data. Yet, by adding new complexity to the specification language.

What all monitoring languages do have in common is, that certain properties are not monitorable. One example is $\Box\Diamond$ `a` that states that `a` must always eventually be satisfied. This property is not monitorable since one can always extend the execution to falsify or satisfy it. The term monitorable is more formally defined by the introduction of *ugly prefixes* [29].

Hierarchy	Example property
Item-level	Each second, the lidar sensor shall produce a value.
System-level	The integral over one second of acceleration readings should reflect the change in velocity within the last second.
Aircraft-level	Whenever a command to increase altitude is given, a corresponding behavior shall be observed by an altimeter reading within one second.
Mission-level	The aircraft shall fullfil its mission to fly to waypoint G, reaching first waypoint A and then waypoint B in between.
Operation-level	Whenever an emergency helicopter enters the airspace, the aircraft shall trigger a safe landing within 30 s.

Table 1 Example temporal monitoring properties for each abstraction level in the monitoring hierarchy.

B. Statistical properties

The previous section used atomic propositions – values that can be either true or false – to describe the properties. For this, a preprocessing step that translates sensor or control values to atomic propositions is required. This is useful for information flow properties like the one presented in Section IV.A, but to get a better understanding and to decide on the health of a sensor, the monitor should work with the actual value. Consider the first property in Table 2 as an example. This property can be described with the following logical formula $\Box$ `altitude_varies_at_most_five_meters`. However, much information is abstracted into the atomic proposition that should be part of the monitor description. Instead of using temporal logic, we use stream-based specification languages as monitor descriptions. More concretely, we use the RTLola framework presented in Section II. In RTLola, the previous example has the following specification:

```
input altitude: Float64
output avg_altitude @1Hz eval with altitude.aggregate(over: 1s, using: avg)
trigger |avg_altitude - avg_altitude.offset(by: -1).defaults(to: avg_altitude)| > 5.0 "Deviation to high!"
```

In this specification, the monitor receives the actual sensor values, illustrated by the input stream `altitude`. It then computes the average over a one-second window and compares the consecutive values of the output stream `avg_altitude`. The output of the monitor can be quantitative, stream values that can be stored in a log-file or visualized in a possible next step. Additionally, trigger streams like the one in Line 3 notify the user every time the property is violated. These streams contain a boolean condition and a message sent if the condition is satisfied.

The previous example shows that RTLola supports different operators to handle the actual sensor values. These are the standard arithmetic and boolean operators and specific operators to argue over time. One of these operators is `aggregate`, which takes a timed window and an aggregation function as arguments besides a reference to a stream it aggregates. The timed window can be either a discrete number or a real-time bound. The aggregation function needs to be a list homomorphism, i.e., a function in which we can split the inputs arbitrarily, apply the function to the subparts, and then combine these parts to handle the real-time window efficiently at runtime. Fortunately, common statistical properties like the average fall into this class of properties and can be computed efficiently with a finite amount of memory. However, statistical properties, such as computing the median over a real-time window, require storing all received values, which would require an infinite amount of memory. In practice, online monitors that evaluate the outcome during flight should be restricted to the former. This does not need to be a limitation for offline log-file analysis purposes. All properties in Table 2 can be evaluated efficiently during flight.

Hierarchy	**Example property**
Item-level	The average altitude over one second should only vary each second by at most five meters.
System-level	The perception module shall receive values by the lidar and camera sensor by a maximum offset of 100 ms.
Aircraft-level	The maximal path deviation of the aircraft within the last 30 s shall be at most ten meters.
Mission-level	During flight, a progress percentage for reaching waypoint A shall be observed.
Operation-level	At least two contingencies must still be possible with the current battery consumption.

Table 2 Example statistical monitoring properties for each abstraction level in the monitoring hierarchy.

C. Parametrized properties

In Section IV.B, we used properties in which the number of evaluations was bounded. However, there are properties for which this assumption does not hold, such as the properties in Table 3. For a concrete example, we use the mission-level property that checks if a waypoint is reached in a specified duration. For this property, we need to compute specific information for each waypoint. However, we do not know the number of waypoints when building the monitor, so the number of stream definitions is not bounded. RTLola supports parametrization for these kinds of properties. These definitions now define a template where an input to the monitor can spawn an instance of the template. The RTLola specification in Listing 1 gives an example formalization of the previously discussed property.

Hierarchy	**Example property**
Item-level	Whenever a specific actuator position is commanded, the corresponding actuator feedback should be received within two seconds.
System-level	Whenever a candidate landing site has been tracked by a perception algorithm, the tracked candidate must be persistently tracked the next three seconds before it is classified as an alternative landing site.
Aircraft-level	Whenever an intruder aircraft is detected, the intruder must be characterized as threat when entering the remain-well-clear volume.
Mission-level	Whenever a new waypoint is added to the waypoint list, this waypoint shall be reached within the specified time bound.
Operation-level	A new polygon that represents a *stay-out* region, where the aircraft is not allowed to fly, is added.

Table 3 Example parametrized monitoring properties for each abstraction level in the monitoring hierarchy.

This specification gets the current position of the vehicle with the input stream `pos`, the position of a new waypoint with `new_wp`, and the duration in which the vehicle should reach this waypoint with `timebound_new_wp_in_secs`. We then decide for each waypoint if this waypoint was reached by computing the distance between the waypoint and the current position of the vehicle. This is described with the **eval** equation in `reached(wp)`, similarly to the specification presented in Section IV.B. Compared to the previous examples, the stream equation now defines stream templates for which the monitor can have several instances. For this, the equation is annotated with a set of parameters – the instance ID– bounded by the stream values defined in the **spawn** equation. The last equation **close** then defines when an instance needs to be closed and is no longer available. Next, we define a template that counts the number of seconds it takes to reach one waypoint and then check if this duration is smaller than the requested duration. Finally, we check for each instance if the condition is satisfied and notify the user otherwise. We again use an aggregation function that now aggregates all instances instead of aggregating values over time. This expressiveness, however, comes with a cost: Because the number of instances is not bounded, neither is the memory or the execution time of the monitor.

```
input pos: (Float32, Float32, Float32)
input new_wp: (Float32, Float32, Float32)
input timebound_new_wp_in_secs: UInt32

output reached(wp) // Computes when a waypoint is reached
  spawn with new_wp
  eval with |pos - wp| < ϵ
  close when reached(wp)

output wp_timer (wp) // Represents the elapsed time since the new waypoint was provided
    spawn with new_wp
    eval @1Hz with wp_timer(wp).offset(by: -1).defaults(to: 0) + 1
    close when reached(wp)

output violation (wp, duration) // Boolean template that checks whether the timer is exceeded
  spawn with (new_wp, timebound_new_wp_in_secs)
  eval with wp_timer(wp) > duration
  close when reached(wp)

trigger violation.aggregate(using: ∃) "WARNING: Waypoint not reached in time."
```

Listing 1 A RTLola specification that uses parametrization to monitor whether a new waypoint is reached within a specified time bound.

D. Spatial properties

Spatial properties allow to specify changes in space given a spatial model. Such a spatial model can be a weighted graph as used by *Spatio-Temporal Reach and Escape Logic (STREL)* [30, 31] where nodes and edges have physical and logical attributes that can change in time. The spatial representation is required since spatial operators explore the possibility of an event. For example, to evaluate the property "there must always be an emergency landing site within one kilometer" requires the current position of the aircraft and the positions of the landing sites. To formalize such a requirement, STREL extends temporal logic by spatial operators that are augmented by elements d of a distance domain and a distance function f that maps paths of the weighted graph to the distance domain. Also, ϕ represent a STREL property that may use temporal operators such as $\square$ and $\Diamond$. The additional spatial operators are

- ϕ_1 $\textbf{reach}^{f}_{[d_1,d_2]}$ ϕ_2 that is satisfied if from a location where ϕ_1 is satisfied, we can reach a location where ϕ_2 is satisfied while following a path in the spatial model such that the distance function f given this path stays within $[d_1, d_2]$,
- $\textbf{escape}^{f}_{[d_1,d_2]}$ ϕ that is satisfied if from the current location we can find a path where the distance function f evaluates to a distance within $[d_1, d_2]$ and forall locations in that path ϕ is satisfied,
- $\textbf{somewhere}^{f}_{\leq d}$ ϕ that is satisfied if there is a location at a distance between d_1 and d_2 where ϕ is satisfied,
- $\textbf{everywhere}^{f}_{\leq d}$ ϕ that is satisfied if all locations within a distance of d_1 and d_2 satisfy ϕ.

Given the spatial model where these nodes represent alternative landing sites and the aircraft, the weights of the edges represent the Euclidean distance in meters, and the distance function *dist* is the sum of weights along a path, the above requirement can be formalized as $\square$ ($\textbf{somewhere}^{dist}_{[0,200]}$(`node = emergency_landing_site`)).

Other spatial properties are given in Table 4. Note that the examples for item-level and system-level introduce spatial properties given a spatial model that represents a network topology where weights on edges indicate the latency. For example, given the *hop* distance function that counts the number of edges along a path, the requirement "the mission manager (mm) must be directly connected to the ground control station (gcs)" can be specified as (node = mm) $\texttt{reach}^{hop}_{[0,1]}$ (node = gcs).

Hierarchy	Example property
Item-level	The output of the sensor must be broadcasted to all subscribers with a latency of at most one second.
System-level	The mission manager must be directly connected to the ground control station.
Aircraft-level	The average noise level within one kilometer of houses should always be less than 50 dB.
Mission-level	There must always be an alternative trajectory leading to an alternative landing site within 200 m.
Operation-level	There shall always be an emergency landing site within one kilometer.

Table 4 Example spatial monitoring properties for each abstraction level in the monitoring hierarchy.

Remark: There is a relationship between the property types. In fact, property types are often combined for a specific property. For instance, statistical properties are often temporal since they are computed over consecutive values and are not only limit to current input values.

V. Discussion of Using Monitoring Specification Languages

This section discusses advantages and disadvantages of using a formal language to specify monitoring properties. Further, a comparison between a specified property and a handwritten monitor is depicted.

Disadvantages

There is no language to rule them all. We have presented three specification languages that complement each other. Currently, there is no language that covers all property types. Further, to provide guarantees such as memory bounds, some specification languages restrict their expressiveness to efficient fragments [23].
Mostly academic tools. There is active research on specification languages, new operators are added and previous operators are changed, which can be hard to keep up. So far, there are only steps towards commercialization of Runtime Monitoring tools based on specification languages, but no industrial tool yet.
Yet another language to learn. Formal methods and specification languages in particular are not part of standard engineering education, which makes it sometimes difficult for engineers to apply such methods. The engineers require clear benefits before investing in learning.
Never change a running system. Often there is legacy code that already contains monitor code. It is an investment to extract this code, especially when the code was already approved.
There are properties that cannot be expressed. Specification languages have a clear syntax that has benefits like an automated analysis of the written specification, but also downsides when the syntax of the language limits expressiveness.

Advantages

Formalization process helps to think about monitoring properties. When formalizing a requirement, the language already guides towards specifying "good" monitoring properties and highlights aspects that might be missing. For instance, properties can be inefficient to monitor or default values may need to be provided for accesses non-existing past or future events.
Specification languages are concise and precise. When familiar with the language, specification are easy to write since the syntax is concise and precise. Especially, during development, when properties change frequently, this helps improve maintenance and also reduces errors when writing code by hand. This is especially important when considering future autonomous aircraft systems that will include more and more black-box components that require monitoring capabilities.
Analysis of specification language. Formal specification languages are designed with analysis in mind. For instance, RTLola is designed to analyze the memory to give bounds on the memory consumption of the generated monitor implementation. Further, specification analysis can help to check consistency and to provide verified guarantees on the behavior of the monitor [22] that help argue ASTM monitor coverage.
Automatic generation of monitor implementation. Monitors can be automatically derived given a specification. Previously

mentioned analysis also provide artifacts that help to safely integrate them. Further, the SW/HW implementation is decoupled from the specification. This means one is not only limited to C-Code or VHDL-Code but can change them on demand. Additionally, optimizations on specification level directly carry over to all implementations.
Closes gap between natural language and implementation. Operators like *always*, *eventually*, or *aggregate(over: 1s, using: avg)* can be directly linked to natural language requirements. Hence, they have clear advantages when tracing them for certification.

Monitoring is part of everyday software. However, monitoring based on a specification language is not yet. The disadvantages show that there are challenges ahead. Most importantly, specification languages need to converge to a stable version, maybe even with an industrial product, before they can be widely used in industry. Yet, the advantages also show that monitoring specification languages have clear benefits especially for future autonomous aircraft systems. These systems involve black-box components that require monitoring complex properties for integrating them safely into the system.

Next, to illustrate the benefits of a monitoring specification language compared to handwritten code. We consider the example monitoring property "when the command to increase altitude is given (i.e., > 0), then within the next second the altitude should be increasing for at least three altimeter readings". This property can be formalized in MLTL as $\square(\texttt{hgt\_cmd}> 0 \rightarrow \Diamond_{[0s,1s]}\ \texttt{increase(alt, 3)})$ where `increase(alt, 3)` is an atomic proposition that is computed outside the specification. The property can also be formalized in RTLola as depicted in Listing 2 where `increase(alt, 3)` is computed within the specification. Line 4 shows how we check if the last three values of `alt` are increasing, i.e., `alt > alt' > alt"` where ' indicates the previous value. Then in Line 5, if we receive an command to increase altitude, we spawn a `watchdog` . The first value of this output is `true` and will be produced upon its first activation that is after `1s`, if the spawned output was not closed before. The close condition is based on the computation in Line 4.

A corresponding handwritten monitor written in pseudo code is depicted in Listing 3 which for simplicity assumes that both inputs arrive synchronously. Here, inputs are read in Line 19 and previous input values are updated correspondingly from Line 21 to Line 32. To track the expiration of the time limit of one second due to a positive altitude command (Line 34), the *obligation* variable is set to *false* and a separate thread is spawned (Lines 35-36) that sleeps for one second and checks whether the *obligation* was satisfied in the meantime (Lines 5-11). The *obligation* represents an increase in altitude over three consecutive altitude readings (Line 39). Given this explanation, it is clear how it matches the monitor property. Yet, without such an explanation, it is quite hard to understand the task of the monitor. In contrast, the specification language offers a more compact representation. Even if one is not familiar with the details of the specification language, the property is more intuitive to understand since it is more compact and directly supports temporal notions. Further, memory management and race conditions of concurrent computations are prone to errors and can be avoided if a dedicated monitoring language is used.

```
// Inputs                                                                                   1
input alt_cmd, alt: Float64, Float64                                                        2
// Outputs                                                                                  3
output increase_alt : Bool := alt.aggregate(by: -2, using: >)                               4
output watchdog_altitude @1Hz spawn when alt_cmd > 0.0 eval with true close when increase_alt 5
trigger watchdog_altitude "Violation"                                                       6
```

Listing 2 An RTLola specification that represents "when the command to increase altitude is given, then within the next second the altitude should be increasing for at least three altimeter readings"

VI. Conclusion

Monitoring autonomous aircraft systems is a challenging task. It involves monitoring complex safety properties that include notions of time and space. Further, system architectures that allow the monitor to access the overall systems are becoming increasingly important, as the monitor needs a holistic perspective to assure the correctness of black-box components such as neural networks. These challenges pose the risk that current monitoring approaches, which rely on handwritten code, cannot keep up. In the paper, we have proposed the use of formal monitoring specification languages to address these challenges. We first presented a hierarchy of monitoring properties. The hierarchy is based on the existing levels of SAE ARP4761, such as item-, system-, and aircraft-level, but is extended to include mission- and operation-level for autonomy. This allows separation of concerns such as "Are the functions of the aircraft working?", "Is the mission that the autonomous aircraft is supposed to fulfill being followed?", and "Is the mission objective within the operational limits?”. Then, we gave a categorization of different types of monitoring properties: temporal, statistical, spatial, and parametrized. We used four specification languages, namely LTL, MLTL, RTLola, and STREL, to formalize example requirements at different levels of the introduced hierarchy. The examples show that requirements can be concisely expressed using dedicated temporal and spatial operators. Yet, there is no single specification language that covers all property types. Further, there is a tradeoff between the number of operators, i.e., the complexity of the syntax, and the assumptions on the system under scrutiny, e.g., synchronous or asynchronous semantics. We then discussed such aspects when we presented the advantages and disadvantage of using a specification language for monitoring. Using an handwritten monitor code example, we have showed how specification languages offer a higher level of abstraction: monitoring properties can be formalized more compact and are easier to write and to understand. Hence, they improve the maintainability of monitoring capabilities that is required to handle the increased complexity of future autonomous aircraft systems.

```
 1  double alt_cmd, alt, alt_past_1, alt_past_2;
 2  bool watchdog_active, obligation;
 3
 4  void timer(){ // Represents the time limit of the watchdog
 5     watchdog_active = true;
 6     sleep(1);
 7     if(!obligation){
 8        print("Violation");
 9     }
10     watchdog_active = false;
11  }
12
13  void monitor() {
14     int default_count = 2;
15      watchdog_active = false;
16     while(1) {
17        //read inputs
18        double tmp_alt = alt;
19        alt_cmd, alt = read_blocking() // Reads inputs and blocks if no inputs are present
20        // updates past values by respecting default values if value does not exist yet.
21        if (default_count == 2){
22           alt_past_1 = alt;
23           alt_past_2 = alt;
24           default_count--;
25        } else if (default_count == 1){
26           alt_past_2 = alt;
27           default_count--;
28        }
29        else {
30           alt_past_2 = alt_past_1;
31           alt_past_1 = tmp_alt;
32        }
33        // activates watchdog if positive altitude command was given.
34        if(alt_cmd > 0 && !watchdog_active){
35           obligation = false;
36           thread watchdog(timer); // Spawn a thread that executes the function timer(), see Line 5
37        }
38        // updates obligation if altitude was increasing for three consecutive altitude readings.
39        obligation = alt > alt_past_1 && alt_past_1 > alt_past_2;
40     }
41  }
```

Listing 3   A handwritten monitor that represents “when the command to increase altitude is given, then within the next second the altitude should be increasing for at least three altimeter readings”

Acknowledgments

This work was partially supported by the Aviation Research Program LuFo of the German Federal Ministry for Economic Affairs and Energy as part of “Volocopter Sicherheitstechnologie zur robusten eVTOL Flugzustandsabsicherung durch formales Monitoring”(No. 20Q1963C). This work was also supported in part by NSF:CPS Award #2038903, NSF:CAREER Award #1664356, and NASA Cooperative Agreement Grant #80NSSC21M0121.

References

[1] Rein, W., "Autonomous Drones: Set To Fly, But May Not Comply; 5 Major Obstacles For Unmanned Aircraft Systems," , 2018. URL https://www.wileyrein.com/newsroom-articles-Autonomous-Drones-Make-It-Easier-to-Fly-But-Harder-to-Comply.html.

[2] Nagarajan, P., Kannan, S. K., Torens, C., Vukas, M. E., and Wilber, G. F., "ASTM F3269 - An Industry Standard on Run Time Assurance for Aircraft Systems," *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, Inc., 2021. URL https://elib.dlr.de/144352/.

[3] for Civil Aviation Equipment, T. E. O., "Guidelines for development of civil aircraft and systems," *EUROCAE ED-79A*, December 2010.

[4] SAE, "ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment - SAE International," , Dec 1969. URL https://www.sae.org/standards/content/arp4761, [Online; accessed 1. Dec. 2022].

[5] Joint Authorities for Rulemaking of Unmanned Systems, "JARUS Guidelines on Specific Operations Risk Assessment (SORA)," , 2019. URL http://jarus-rpas.org/sites/jarus-rpas.org/files/jar_doc_06_jarus_sora_v2.0.pdf.

[6] Fisher, M., Mascardi, V., Rozier, K. Y., Schlingloff, H., Winikoff, M., and Yorke-Smith, N., "Towards a Framework for Certification of Reliable Autonomous Systems," *20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Springer, 2021.

[7] European Union Aviation Safety Agency (EASA), "Artificial Intelligence Roadmap, A Human-Centric Approach to AI in Aviation, Version 1.0," https://www.easa.europa.eu/newsroom-and-events/news/easa-artificial-intelligence-roadmap-10-published, 2020.

[8] European Union Aviation Safety Agency (EASA), "Concepts of Design Assurance for Neural Networks (CoDANN)," https://www.easa.europa.eu/sites/default/files/dfu/EASA-DDLN-Concepts-of-Design-Assurance-for-Neural-Networks-CoDANN.pdf, 2020.

[9] European Union Aviation Safety Agency (EASA), "Concepts of Design Assurance for Neural Networks (CoDANN) II," https://www.easa.europa.eu/document-library/general-publications/concepts-design-assurance-neural-networks-codann-ii, 2021.

[10] European Union Aviation Safety Agency (EASA), "Concept Paper First Usable Guidance for Level 1 Machine Learning Applications," https://www.easa.europa.eu/easa-concept-paper-first-usable-guidance-level-1-machine-learning-applications-proposed-issue-01pdf, 2021.

[11] Torens, C., Juenger, F., Schirmer, S., Schopferer, S., Zhukov, D., and Dauer, J. C., "Safe Autonomy for Urban Air Mobility," *AIAA SCITECH 2023 Forum*, American Institute of Aeronautics and Astronautics, 2023.

[12] Falcone, Y., Krstić, S., Reger, G., and Traytel, D., "A taxonomy for classifying runtime verification tools," *International Journal on Software Tools for Technology Transfer*, Vol. 23, No. 2, 2021, pp. 255–284.

[13] Schumann, J., Rozier, K. Y., Reinbacher, T., Mengshoel, O. J., Mbaya, T., and Ippolito, C., "Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems," *International Journal of Prognostics and Health Management*, Vol. 6, No. 1, 2015.

[14] Rozier, K. Y., and Schumann, J., "R2U2: Tool Overview," *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, Kalpa Publications in Computing, Vol. 3, edited by G. Reger and K. Havelund, EasyChair, 2017, pp. 138–156. https://doi.org/10.29007/5pch, URL https://doi.org/10.29007/5pch.

[15] Geist, J., Rozier, K. Y., and Schumann, J., "Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems," *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, Lecture Notes in Computer Science, Vol. 8734, edited by B. Bonakdarpour and S. A. Smolka, Springer, 2014, pp. 215–230. https://doi.org/10.1007/978-3-319-11164-3_18, URL https://doi.org/10.1007/978-3-319-11164-3_18.

[16] Kempa, B., Zhang, P., Jones, P. H., Zambreno, J., and Rozier, K. Y., "Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2," *Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, Lecture Notes in Computer Science (LNCS), Vol. 12288, Springer, Vienna, Austria, 2020, pp. 196–214. https://doi.org/10.1007/978-3-030-57628-8_12, URL http://research.temporallogic.org/papers/KZJZR20.pdf.

[17] Kempa, B., Johannsen, C., and Rozier, K. Y., “Improving Usability and Trust in Real-Time Verification of a Large-Scale Complex Safety-Critical System,” *Ada User Journal*, Vol. September, 2022.

[18] European Union Aviation Safety Agency (EASA), “Specific Category - Civil Drones,” , 2022. URL https://www.easa.europa.eu/domains/civil-drones/drones-regulatory-framework-background/specific-category-civil-drones.

[19] Torens, C., Nikodem, F., Dauer, J., Schirmer, S., and Dittrich, J. S., “Geofencing requirements for onboard safe operation monitoring,” *CEAS Aeronautical Journal*, 2020. URL https://elib.dlr.de/135054/.

[20] Schirmer, S., and Torens, C., *Safe Operation Monitoring for Specific Category Unmanned Aircraft*, Springer International Publishing, Cham, 2022, pp. 393–419. https://doi.org/10.1007/978-3-030-83144-8_16, URL https://elib.dlr.de/145080/.

[21] Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., and Torens, C., “RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft,” *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, Lecture Notes in Computer Science, Vol. 12225, edited by S. K. Lahiri and C. Wang, Springer, 2020, pp. 28–39. https://doi.org/10.1007/978-3-030-53291-8_3, URL https://doi.org/10.1007/978-3-030-53291-8_3.

[22] Dauer, J. C., Finkbeiner, B., and Schirmer, S., “Monitoring with Verified Guarantees,” *CoRR*, Vol. abs/2110.11755, 2021. URL https://arxiv.org/abs/2110.11755.

[23] Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., and Torfah, H., “StreamLAB: Stream-based Monitoring of Cyber-Physical Systems,” *Computer Aided Verification*, edited by I. Dillig and S. Tasiran, Springer International Publishing, Cham, 2019, pp. 421–431.

[24] Finkbeiner, B., Oswald, S., Passing, N., and Schwenger, M., “Verified Rust Monitors for Lola Specifications,” *Runtime Verification*, edited by J. Deshmukh and D. Ničković, Springer International Publishing, Cham, 2020, pp. 431–450.

[25] Baumeister, J., Finkbeiner, B., Schwenger, M., and Torfah, H., “FPGA Stream-Monitoring of Real-Time Properties,” *ACM Trans. Embed. Comput. Syst.*, Vol. 18, No. 5s, 2019. https://doi.org/10.1145/3358220, URL https://doi.org/10.1145/3358220.

[26] Fainekos, G. E., Sankaranarayanan, S., Ueda, K., and Yazarel, H., “Verification of automotive control applications using S-TaLiRo,” *2012 American Control Conference (ACC)*, 2012, pp. 3567–3572. https://doi.org/10.1109/ACC.2012.6315384.

[27] Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S., “S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems,” *Tools and Algorithms for the Construction and Analysis of Systems*, edited by P. A. Abdulla and K. R. M. Leino, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 254–257.

[28] Jin, X., Donzé, A., Deshmukh, J. V., and Seshia, S. A., “Mining Requirements from Closed-Loop Control Models,” *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, Association for Computing Machinery, New York, NY, USA, 2013, p. 43–52. https://doi.org/10.1145/2461328.2461337, URL https://doi.org/10.1145/2461328.2461337.

[29] Bauer, A., Leucker, M., and Schallhart, C., “Runtime verification for LTL and TLTL,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 20, No. 4, 2011, pp. 1–64.

[30] Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L., and Silvetti, S., “MoonLight: A Lightweight Tool for Monitoring Spatio-Temporal Properties,” *Runtime Verification*, edited by J. Deshmukh and D. Ničković, Springer International Publishing, Cham, 2020, pp. 417–428.

[31] Loreti, M., Bortolussi, L., Bartocci, E., and Nenzi, L., “A Logic for Monitoring Dynamic Networks of Spatially-distributed Cyber-Physical Systems,” *Logical Methods in Computer Science*, Vol. 18, 2022.