

# Monitoring Unmanned Aircraft: Specification, Integration, and Lessons-Learned

Jan Baumeister[1(✉)], Bernd Finkbeiner[1], Florian Kohn[1], Florian Löhr[2], Guido Manfredi[2], Sebastian Schirmer[3], and Christoph Torens[3]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{jan.baumeister,finkbeiner,florian.kohn}@cispa.de
[2] Volocopter GmbH, Bruchsal, Germany
{florian.loehr,guido.manfredi}@volocopter.com
[3] German Aerospace Center (DLR), Cologne, Germany
{sebastian.schirmer,christoph.torens}@dlr.de

**Abstract.** This paper reports on the integration of runtime monitoring into fully-electric aircraft designed by Volocopter, a German aircraft manufacturer of electric multi-rotor helicopters. The runtime monitor recognizes hazardous situations and system faults. Since the correct operation of the monitor is critical for the safety of the aircraft, the development of the monitor must follow strict aeronautical standards. This includes the integration of the monitor into different development environments, such as log-file analysis, hardware/software-in-the-loop testing, and test flights. We have used the stream-based monitoring framework RTLola to generate monitors for a range of requirements. In this paper, we present representative monitoring specifications and our lessons learned from integrating the generated monitors. Our main finding is that the specification and the integration need to be decoupled, because the specification remains stable throughout the development process, whereas the different development stages require a separate integration of the monitor into each environment. We achieve this decoupling with a novel abstraction layer in the monitoring framework that adapts the monitor to each environment without affecting the core component generated from the specification. The decoupling of the integration has also allowed us to react quickly to the frequent changes in the hardware and software environment of the monitor due to the fast-paced development of the aircraft in a startup company.

**Keywords:** Runtime Verification · Stream Monitoring · Autonomous Aircraft

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).

© The Author(s) 2024
A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 207–218, 2024.
https://doi.org/10.1007/978-3-031-65630-9_10

## 1 Introduction

The new generation of fully-electric aircraft pioneered by companies like Volocopter promises a revolution in urban air mobility. Fully-electric aircraft air taxis, cargo drones, and longer-range passenger aircraft will provide transit solutions that are emission-free and thus more sustainable and efficient than traditional forms of air transport. A critical part of the safety engineering of such aircraft is to analyze log-files and tests, as well as the real-time data obtained during the actual flight, so that the health status of the system can be assessed and mitigation procedures can be initiated when needed. In this paper, we report on the design and integration of formally specified monitors into aircraft developed by Volocopter, based on the monitoring framework RTLola. The goal of our collaboration over the past three years has been to explore the benefits and challenges of applying formal runtime verification within the strict aeronautical standards of aircraft development.

Volocopter specializes in the design, manufacturing, and operations of electric Vertical Takeoff and Landing (eVTOL) vehicles. The company targets Urban Air Mobility (UAM) operations, i.e., passenger and cargo transportation above and around cities. These operations involve high population density on the ground and high traffic density in the air. Consequently, all developments must meet the highest level of safety similar to airliners: one failure for every billion hours flown. To ensure such a level of safety, the design of the vehicles follows aeronautical standards, especially SAE's ARP4754b [14] to ensure the coherency between the concept of operation, requirements, design, and implementation. The development cycle described in this standard uses a layered approach with multiple verification and validation steps.

RTLola [3,8] is a formal monitoring framework that consists of a stream-based specification language for real-time properties, an interpreter, and compilers into software- and hardware-based execution platforms. An RTLola specification of hazardous situations and system failures is statically analyzed in terms of consistency and resource usage and then automatically translated into an FPGA-based monitor. This approach leads to highly efficient, parallelized monitors with formal guarantees on the noninterference of the monitor with the normal operation of the monitored system.

Previous case studies with RTLola [2] and similar frameworks, such as R2U2 [10] and Copilot [12], have already shown that properties that are critical for the safety of the aircraft can readily be expressed in such formal languages and that the resulting monitors can be integrated into real systems. Our ambition has been to go beyond such one-time applications, and integrate the specified monitors into the complete development process. This means that the generated monitors are not only integrated into the specific setup of the case study, but rather are continuously adapted according to the needs of the development process.

We consider monitoring in all stages of the development process. Initially, the role of the monitor is to annotate log-files and guide the user during an offline analysis, e.g. these annotations split a test flight into flight-phases for

separate inspection. Next, the monitor validates data from test-benches that check that external components conform to their specifications, such as delivering data within deadlines. Finally, the monitor validates safety requirements during test flights. The monitoring specifications are based on the requirements of the various regulatory authorities and cover a range of safety-critical requirements from single-component checks to system-level health.

Our main finding is that the specification and the integration need to be decoupled, because the specification remains stable throughout the development process, whereas the different development stages require a separate integration of the monitor into each environment. We achieve this decoupling with a novel abstraction layer in the monitoring framework that adapts the monitor to each environment without affecting the core component generated from the specification. In the abstraction layer, the monitor is framed with two new components, the *event conversion* and the *verdict converison*. The decoupling of the integration has also allowed us to react quickly to the frequent changes in the hardware and software environment of the monitor due to the fast-paced development of the aircraft in a startup company.

### 1.1 Related Work

Runtime monitoring is a scalable dynamic verification approach that has been applied to a variety of domains [9,11]. For cyber-physical systems, many monitoring tools exist [1,2,12], but despite integration being an important part of the usage of monitoring [7], tools are often specific to certain environments and leave embedding in different environments to the user, i.e., the user needs to establish a connection, parse received events, and forward it to the monitor. For some specific environments, these user efforts are reduced. For instance, SOTER [4] a specification language that is based on the P language [5], was recently extended [16] to produce code for the Robot Operating System (ROS), which allows to just specify which ROS topics are subscribed and published. Similarly, TeSSLa features keywords to subscribe and publish ROS topics [16]. A more generic approach is pursued by R2U2 Version 3.0 [10] which allows to specify C-like structs. This makes it easy for engineers to receive structs and just forward them to the monitor unit. In this work, we foster this kind of generalization by providing an automatic mapping of the received and the forwarded events.

## 2 Stream-Based Monitoring

RTLola is a real-time monitoring framework [8] aimed at, but not exclusively applicable to, cyber-physical systems. At its core is a stream-based specification language that distinguishes between two kinds of streams: Input streams represent sensor readings from the system under observation. Output streams perform computations over these input streams and other output streams. Special kinds of output streams, called triggers, define violations based on boolean conditions.

Equipped with a message, they notify the system operator when a violation is detected. Consider the following example:

```
input altitude: Float
output average_alt @1Hz := altitude.aggregate(over: 60s, using: avg).defaults(to: 0.0)
trigger average_alt > 300.0
```

In this example, the monitor observes the altitude of the system through the input stream `altitude`. The output stream `average_alt` aggregates all values of this input stream over the last minute and computes the average of these values. It also highlights the real-time capabilities of RTLola. By explicitly annotating the output stream with a frequency, the monitor cannot only react to events but also proactively perform computations. More concretely, the output stream evaluates at a fixed frequency of `1 Hz`. The final defined trigger then notifies an operator if the average altitude is above 300.

## 3 Setup

All components that are integrated into aircrafts designed by Volocopter need to follow aeronautical standards, especially SAE's ARP4754b [14]. This standard ensures that the concept of operation, requirements, design, and implementation are coherent. In general, it describes a development cycle using a layered approach with multiple verification and validation steps, i.e., new components are validated in different environments that get closer to the operation with each step.

Integrating a monitor in this setup is two-folded: 1. the monitor verifies the behavior of new components and 2. the monitor itself needs to be verified The monitor can provide valuable feedback when new components undergo the aforementioned validation steps. This feedback includes statistical assessments or violations of given requirements. Yet, the monitor as a safety-critical component needs to be evaluated in the same manner.

### 3.1 Monitoring Applications

This section presents four applications that highlight the benefit of the monitors feedback during the development of new components:

1. *Debugging.* A monitor is developed alongside the component giving full information about its internal state. During the execution of the system, the monitor checks whether the component works as intended by the developer used as a white-box testing component.
2. *Validation.* The monitor is developed independently of the component and checks its behavior based on the inputs and outputs of defined test cases. Hence, the monitor functions as a black-box testing component. The monitor output on these test cases is then used as a report for internal validation, validation of components by external companies, or as proof of conformity for aviation authorities.

3. *Pre-Post-Flight Analysis.* Before the flight, the monitor checks whether all necessary components are operational. After the flight, the monitor computes more sophisticated information to better evaluate the flight and detect irregularities that were not detected during the flight.
4. *In-Flight Analysis/Safe Integration.* The monitor communicates with the remote operator. e.g., through the User Interface of the ground control station, to provide feedback about the safety of the drone. It validates the correctness of individual components to ensure a safe flight or monitors the flight operation. For instance, it supports the pilot by checking that the drone stays within safe flight parameters such as a geofence.

Before presenting concrete specifications for each application in Sect. 5, we elaborate on the validation of a monitor.

### 3.2 Development Cycle for the Monitor

This section introduces the four environments into which the monitor must be integrated to validate its correctness.

1. *Log-File Analysis.* This step evaluates the functional correctness of the specification. We test the generated monitors against traces that violate or satisfy the specification and analyze the output of the monitor.
2. *Software-in-the-Loop (SiL).* The monitor interacts with simulated systems and environments. This step is crucial for a runtime monitor since most temporal behaviors are not visible until these tests.
3. *Hardware-in-the-loop (HiL).* This step is similar to the SiL environment. However, the monitor and the system run on the actual resource-constrained hardware used in the aircraft. This setup brings even more time-related effects to the evaluation and allows an evaluation with replayed flight data.
4. *Flight Testing.* Running the monitor in parallel with the flying aircraft allows for assessing the impact of all effects coming from the aircraft, the ground system, and the environment.

The integration of the monitors in the different validation environments poses new challenges for the monitoring framework. In our experience, each step in the development process relies on different ways of communication. For instance, in the log-file analysis, events are processed in CSV-format, while during test flights, the communication with the monitor uses a custom protocol over TCP. Yet, the changes in the monitor should be as minimal as possible to simplify its validation. Specifically, the specification has to remain unchanged after the *Log-file Analysis* as otherwise its functional correctness is not guaranteed anymore.

## 4 Abstract Integration

In this section, we present our approach integrating the RTLola framework [8] into the different environments described in Sect. 3.

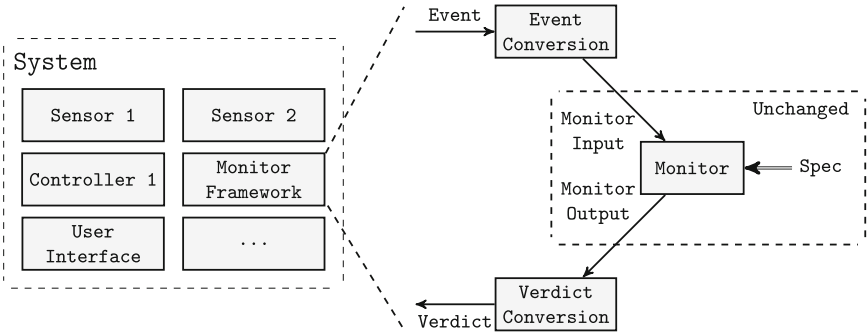



**Fig. 1.** Overview of the Generalization

Figure 1 shows an overview of this approach. The system on the left side represents the UAV under development. From a monitoring perspective, the current step in the development cycle does not influence the underlying monitor, only its integration into the system. This is depicted on the right side of Fig. 1. The monitor framework receives or requests incoming data from the different components of the system (`Event`), analyses this data, and produces an output (`Verdict`). In the center of the monitoring framework is a fixed monitor generated from a formal specification. This monitor has a fixed representation of the `Monitor Input` and `Monitor Output` that are independent of the integration.

To bridge the communication gap between the system and the monitor, we propose an abstraction layer that translates system outputs to monitor inputs and vice versa. This abstraction also generalizes the monitor's interface such that no expert knowledge about the concrete monitor is necessary to integrate the monitor. This abstraction layer can vary depending on the specific integration into the development cycle but allows the monitor to remain unchanged following the idea of decoupling the specification from the integration.

Our approach introduces two translation components: the `Event Conversion` and the `Verdict Conversion`. Each component is again split into two parts: *data-acquisition/data-dispatch* and *data-conversion* to allow a generic implementation of the *data-acquisition/data-dispatch*. This results in reusable and maintainable implementation while keeping the changes between the development stages minimal.

In the following, we elaborate on the data-acquisition and data-conversion of the *Event Conversion* in more detail. The results are transferable to the *Verdict Conversion*.

### 4.1 Common Interfaces

The `Event Conversion` is a generic translation layer between the systems output and the monitor input. During instantiation it validates the mapping between the system output and the monitor input, representing the input streams in the

specification. Hence, it checks that the `Events` are a superset of the `Monitor Input` avoiding any runtime errors resulting from an invalid mapping. The *data-acquisition* part of the `Event Conversion` is handled through the *Event Source* interface, while the *data-conversion* is handled by an *Event Factory*. Both interfaces are defined in Fig. 2.

```
pub trait EventSource {
  fn next_event(&mut self)
    -> Result<Event, Error>;
}
```

(a) Event Source Interface

```
pub trait EventFactory {
  fn new(map: &InputStreams, cfg: Config)
    -> Result<Self, Error>;
  fn create(&mut self, ev: Event)
    -> Result<MonitorInput, Error>;
}
```

(b) Event Factory Interface

**Fig. 2.** Common interfaces for the Event Conversion.

The *Event Source* consists of a single function called `next_event`. It is used to communicate to the system that the monitor is ready to accept the next event. The *Event Factory* as a counterpart has two functions: The `new` function gets a description of the input streams derived from the specification and the configuration of the *Event Source*. It then checks if each input in the specification can be matched with the data provided by the `Event Factory` implementation. If successful, it computes a static mapping for each input stream to a data segment in an incoming `Event`. The second function `create` is called for every `Event` and creates the internal event structure `Monitor Input`, given the input mapping.

*Implementation.* We implemented the approach from this section in the RTLola framework and were able to provide implementations for a variety of *Event Sources* that are independent of the data format they receive. These include basic file-based input methods such as reading from stdin or a local file, up to network protocols that receive data over UDP, TCP, or MQTT. We also provide ready-to-use *Event Factories* to parse, for example, data in CSV or PCAP format as well as a binary data parser derived from a user-provided configuration.

Yet, implementing a custom *Event Factory* still requires knowledge about the structure of the `Monitor Input` that is undesired for a successful integration in real-production where implementations need to be maintained by non-monitoring experts. In the RTLola framework, we provide further abstractions over the interfaces presented in Fig. 2 to reduce the required knowledge about the monitoring framework. These abstractions range from helper implementations encapsulating common functionality to procedural macros that automatically generate implementations

```
#[derive(ValueFactory)]
#[factory(prefix)]
struct GPS {
    lat: Float64,
    lon: Float64
}
```

**Fig. 3.** Interfacing a custom data structure.

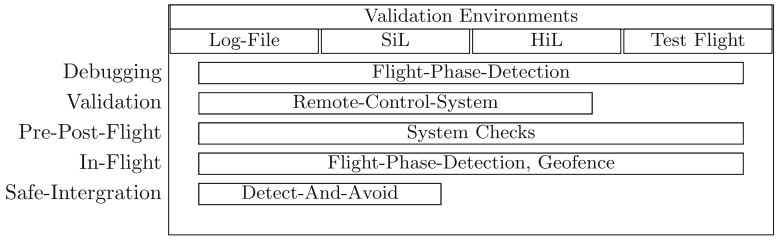



**Fig. 4.** Overview of the concrete integrations that have been performed in the research project

of these interfaces. Figure 3 shows an example of the macro application. It demonstrates a simplified version of a GPS-Package, exposing the fields of the struct to input streams named `GPS_lat` and `GPS_lon`.

## 5 Concrete Integration of Representative Specifications

This section provides a set of representative specifications to validate our approach presented in Sect. 4. The specifications have been obtained by collaborating with flight engineers or from official RTCA [13] standards and cover all monitoring applications.

Figure 4 provides an overview of the concrete specifications and the integration of the generated monitors. The x-axis in this graph visualizes the different environments in which the monitor was integrated, labeled at the top of the figure. The y-axis of the graph in Fig. 4 maps the specifications to their application from Sect. 3.1.

In our setup, log-files are usually given in the csv-format, the SiL is a Matlab Simulation or a simplified replay of log-files and the Hil is a concrete replay of the simulated data or flights on the actual hardware. The test flights were performed on the VoloDrone, a cargo transportation drone that offers highly automated flights with a range of 40km and a payload of up to 200kg. Not all monitors could be validated up to a flight test, but we validated our approach in at least two environments presented in Sect. 3.2. In our experience, the provided *Event Sources* are sufficient for all environments. We either used the *Event Source* to access data stored in files or used the implementations for the different network protocols from Sect. 4. For the concrete implementation of the *Event Factories*, we either use the ready-to-use *CSV Event Factory* or create new implementations using the macros shown in Fig. 3. These implementations do not require any internal knowledge of the monitoring tool as intended by our approach.

We published the specifications on Github[1] after replacing some sensitive information, e.g., by replacing some streams with arbitrary constants.

The rest of the section describes the general idea of the specifications and refers to the concrete monitoring application.

[1] https://github.com/reactive-systems/rtlola-uav-specifications.

```
input rpm : Int64
input src : UInt8
output rpm_1 eval when src == ROTOR_1 with abs(rpm)
output rpm_2 eval when src == ROTOR_2 with abs(rpm)
...
output rpm_on_check := avg(rpm_1.hold(or: 0), rpm_2.hold(or: 0), ...) > ϵ_rpm_on
output rpm_on @1s := rpm_on_check.aggregate(over: 1s, using: avg, or: 0.0) > ϵ_rpm_on_per
...
output phase_1 := ¬take_off ∧ ¬landed ∧ rpm_on ∧ ¬rpm_in_air
```

**Fig. 5.** Excerpt of the specification of the Flight-Phase-Detection

```
/// Property 1: Log message increment
output valid_seq_number := seq_number = seq_number.offset(by: -1, or: -1) + 1
/// Property 7: RC fallback test
output main_fallback_valid_dyn
    spawn when lost_connection_to_master
    close when switch_to_secondary ∨ both_rc_disconnected
    eval @200ms with false
output main_fallback_valid @true := main_fallback_valid_dyn.hold(or: true)
```

**Fig. 6.** Excerpt of the requirements specific to one RCC

*Flight-Phase-Detection (FPD).* The FPD specification detects different flight phases, helping the debugging of correct automated flights. In the log-file analysis, the monitor annotates previous test flights pointing the engineer to critical points, e.g., when no clear phase could be detected. In the software and hardware simulation, we evaluate the handling of asynchronous inputs and the timing of the monitor. For a final flight test, the monitor was integrated into the ground station to check if a flight phase is always detected moving the monitor also to the in-flight application.

Figure 5 presents partially the specification for the FPD. It gets data from several sensors and computes binary flags describing the current state of the drone. One example is the `rpm_one_check` flag that compares the average rotations per minute of all rotors against a threshold. In general, a simple state machine then decides based on these flags if a flight phase is detected and which one. However, the data of the sensors arrives asynchronously with different frequencies and we need to synchronize the flags for the comparison. For this synchronization, the streams `rpm_on` aggregate over the corresponding flags, computing the percentage of how often the condition is satisfied during the last second. This value is then used in `phase_1` stream for the flight phase detection instead of asynchronous `rpm_one_check` flag.

*Remote-Control System (RCS).* Assuring a safe development is especially challenging when combining in-house products with commercial off-the-shelf hardware or software products. In our example, we validated the correctness of an RCS that receives flight commands from different sources and dependent on the configuration decides which source should be used by the system. More concretely, we used RTLola to validate that the requirements given to the company developing the RCS are satisfied by the resulting product. Besides the in-house

validation, this approach comes with certification evidence that can be submitted to the safety agency for the certification process.

As a redundant system, the RCS runs several instances of remote control computers (RCCs) and unions their output. Figure 6 presents some stream-declarations for requirements validating each RCC individually. This specification includes checks of simple invariants such as property one that validates if the sequence number increments, but also includes complex real-time properties exemplified with property seven. The stream declarations for this property implement a watchdog. It reports a violation in the case that the connection to the main controller is lost and the RCC does not switch from the main to the secondary controller in a time frame of 200ms.

*System Checks.* We developed a specification to validate the system parameters of different sensors. This specification included requirements monitoring the battery level and voltage drops, pre-flight sensor inconsistencies, and accelerations bound. Due to the sensible information of this specification that requires knowledge of the complete system, we cannot publish this specification.

*Geofence.* Defining a geography volume and a contingency volume, where the UAV will operate and can be used to maneuver in case of a problem, is part of a risk assessment required for a flight permit in the specific category [6]. This risk assessment also requires a runtime validation that the position of the UAV is within these bounds for which we use the monitors generated from an RTLola specification. Similar to the FPD, we integrated the monitor in the ground station to communicate with the remote pilot. We used the geofence specification from previous case-studies with RTLola [2,15] describing the intersection between the flight vehicle line and the geofence polygon. Further, this paper extends the specification to predict a possible breach of the geofence by computing the minimum distance to each polygon line and to approximate the time until that breach.

*Detect-And-Avoid (DAA).* We use the validation of the DAA function as a representative specification for the safe integration monitoring application. This function is essential for any UAV flying beyond visual line of sight and ensures that the UAV avoids any collision with the surrounding traffic. One of the most common sensors in commercial aviation is the ADS-B in receiver which can sense all surrounding aircraft equipped with ADS-B out emitters. However, this sensor is susceptible to attacks by spoofing, so the RTCA standard [13] demands a safe integration in which this sensor needs to be supported by a secondary sensor, usually an "active surveillance" sensor. Instead of merging both signals, it is common practice to use the active surveillance sensor data to check if parts or all of the ADS-B in signal have been compromised. The challenges for the RTLola specification are similar to the flight phase detection. The specification compares data from sensors with different frequencies and validates these frequencies. Compared to the FPD, the standard assumes in its validation these

frequencies, so a comparison with the last values is sufficient instead of aggregating the data.

## 6 Conclusion

This paper presents the results of our research project investigating the use of runtime monitors implemented in the RTLola framework for the development of unmanned aircraft systems. We demonstrate the benefits of decoupling the specification and integration when the monitor has to undergo the same development as other safety-critical components, in this safety-critical environment. To keep the changes for the monitor during the development as minimal as possible, we presented an abstraction for monitoring frameworks. This abstraction introduces two layers that translate between system outputs and monitoring inputs and vice versa. We conducted a large case study to validate our approach and presented representative specifications for different monitoring applications derived from aeronautical safety standards and internal requirements from Volocopter. In a final step, we performed a test flight where the monitor reported its feedback to the ground control station used by the remote pilot. From a monitoring perspective, this approach can be used to start the development of automatic contingencies triggered by the monitor instead of notifying the pilot.

## References

1. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
2. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RTLola cleared for take-off: monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 28–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_3
3. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. CoRR abs/ arXiv: 2003.12477 (2020)
4. Desai, A., Ghosh, S., Seshia, S.A., Shankar, N., Tiwari, A.: SOTER: a runtime assurance framework for programming safe robotics systems. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, 24-27 June 2019, pp. 138–150. IEEE (2019) https://doi.org/10.1109/DSN.2019.00027
5. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16-19 June 2013, pp. 321–332. ACM (2013) https://doi.org/10.1145/2491956.2462184
6. European Union Aviation Safety Agency (EASA): Specific operations risk assessment (sora) (2019). https://www.easa.europa.eu/en/domains/civil-drones-rpas/specific-category-civil-drones/specific-operations-risk-assessment-sora

7. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021). https://doi.org/10.1007/S10009-021-00609-Z
8. Faymonville, P., et al.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
9. Henzinger, T.A., Karimi, M., Kueffner, K., Mallik, K.: Monitoring algorithmic fairness. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 358–382. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_17
10. Johannsen, C., Jones, P., Kempa, B., Rozier, K.Y., Zhang, P.: R2u2 version 3.0: Reimagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 483–497. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_23
11. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for markov decision processes. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 553–576. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_26
12. Perez, I., Dedden, F., Goodloe, A.: Copilot 3. Tech. rep. (2020). https://ntrs.nasa.gov/citations/20200003164
13. Radio Technical Commission for Aeronautics (RTCA): Minimum operational performance standards (mops) for detect and avoid (daa) systems (2022). https://my.rtca.org/productdetails?id=a1B36000003FXGyEAO
14. S-18 Aircraft and Sys Dev and Safety Assessment Committee: Guidelines for development of civil aircraft and systems arp4754b (2023). https://doi.org/10.4271/ARP4754B
15. Schirmer, S., Torens, C.: Safe Operation Monitoring for Specific Category Unmanned Aircraft, pp. 393–419. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-83144-8_16
16. Shivakumar, S., Torfah, H., Desai, A., Seshia, S.A.: Soter on ros: a run-time assurance framework on the robot operating system. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification, pp. 184–194. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_10

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

