Certification Aspects of Runtime Assurance for Urban Air Mobility

Christoph Torens*, Pranav Nagarajan†, Sebastian Schirmer‡, Johann C. Dauer§
German Aerospace Center (DLR), Department Unmanned Aircraft, Braunschweig, 38102

Jan Baumeister, Florian Kohn, Bernd Finkbeiner
CISPA Helmholtz Center for Information Security, Saarbrücken, 66123

Florian Löhr¶, Guido Manfredi‖
Volocopter GmbH, Bruchsal, 76646

The transition towards autonomous operations for Urban Air Mobility introduces significant safety challenges, necessitating novel safety assurance strategies. One such strategy is runtime assurance, which ensures the safe behavior of a system during its actual operation. This can be implemented by using a safety monitor that detects unsafe behaviors and then activates a switch to a recovery function to return the system to a safe state. This paper investigates the certification aspects of runtime monitoring, a core component of runtime assurance. We analyze the regulatory framework of urban air mobility, and discuss implications of aviation software standards such as DO-178C and it supplements on runtime assurance. As a concrete example to discuss, Detect-and-Avoid is introduced and motivated from the requirements of the Minimum Operational Performance Standards. The use case is analyzed from a system and a software perspective. From a system perspective, the architecture is compared to the runtime assurance standard practice published by ASTM International. From a software perspective, we assess the stream-based specification language RTLola against the development assurance objectives in the de-facto software development standard DO-178C. As an example, we highlight the role of traceability between the different levels of software requirements. The goal of this research is to illustrate the use of runtime monitoring in the context of certification for Urban Air Mobility applications to improve operational safety and enable increasing levels of automation.

I. Introduction

The new aviation segment of Urban Air Mobility (UAM) promises to revolutionize the transportation landscape by offering a sustainable, efficient and more personal form of mobility to the masses. One of the key drivers of this development is the integration of autonomy into UAM systems, which promises significant cost reductions by eliminating the need for onboard human pilots and enhancing the scalability of air taxi services. Nevertheless, the move towards autonomy introduces substantial safety and certification challenges. The absence of a human pilot, conventionally the last line of defense for managing in-flight contingencies, requires the implementation of an alternative safety layer. This layer must be capable of emulating and preferably surpassing the vigilance and responsive capabilities traditionally associated with human oversight. Runtime Assurance (RTA) is a promising architectural framework for implementing such a layer. An example RTA architecture could provide a safety monitor that supervises a complex function and switches control to an appropriate recovery function in an unsafe situation based on a decision logic.

Therefore, the majority of the certification burden in an RTA framework is allocated to the assurance of the safety monitor and the decision logic. This paper focuses on the certification of the safety monitor, leaving the certification of the decision logic for future investigation. We examine the certification of monitors generated from the real-time stream-based specification language RTLola [*]. RTLola allows to specify stream equations that translate timed and

*Researcher, Institute of Flight Systems, Braunschweig, Germany, AIAA Senior Member.
†Researcher, Institute of Flight Systems, Braunschweig, Germany, AIAA Senior Member.
‡Researcher, Institute of Flight Systems, Braunschweig, Germany, AIAA Member.
§Department Head Unmanned Aircraft, Institute of Flight Systems, Braunschweig, Germany, AIAA Member.
¶Senior Project Manager
‖Autonomous Flight Researcher
*https://www.rtlola.org

asynchronously arriving input data, referred to as input streams, to output streams. Note that output streams can be reused by other output streams. Further, RTLola supports temporal operators to reason over time sequences of input events, which can describe safety properties in a more precise and compact way compared to hand-written code. This is particularly useful when observing complex functions such as machine-learning functions for which assurance can not be statically provided and monitoring system states and environmental conditions are a way to verify its behavior during execution. Besides that RTLola supports the specification of temporal properties, the corresponding framework also performs various static analyses to provide reliable guarantees, e.g., memory consumption, prior to execution. It then either directly executes the specification with the RTLola-Interpreter or compiles it to software or hardware. The framework has been successfully tested in the context of monitoring unmanned aerial vehicles (UAVs) before [1]. In this paper, we developed a new compilation of an RTLola specification focusing on the certification of the resulting code. We investigate the nuances of integrating monitoring for runtime assurance into UAM, the challenges of traceability, the regulatory context, and the potential of monitoring to support the certification processes. We explore a path for effectively incorporating RTLola for runtime assurance into UAM systems, thereby providing a critical safety layer and contributing to the progression of the UAM industry.

Related Work

NASA conducted a significant amount of research on runtime assurance as an alternative to development assurance [2] and specifically for safety assurance of urban air mobility [3]. The concept of runtime assurance dates back to Sha et al [4] that introduced the simplex architecture to safeguard an experimental controller. Since then, it has been used for aircraft taxiing [5], safe operation monitoring for unmanned aircraft [6], and other use cases [7, 8]. In 2021, a revised edition of an industry standard on runtime assurance was published by ASTM International, referred to as F3269-21 [9], with the authors providing supplemental context behind the work in a separate publication [10].

Safety monitors are an essential part of such an architecture, but are also extremely important for engineering safety-critical avionics systems in general. Hence, there are different recommended practices to develop such a monitor. For example, ARP-6539 [†] which describes the "Validation and Verification Process Steps for Monitors Development in Complex Flight Control and Related Systems". At the core of this standard is a monitor performance analysis that suggests important parameters for a monitor specification. For instance, a monitor should be robust to nuisance trips and account for reaction and fault confirmation times to ensure the safe activation of fault counter-measures. In this paper, the focus is not on specifying monitoring properties, but rather examine how objectives of a software development assurance process can be fulfilled automatically. Not that system development in aerospace generally needs to comply to certain standards like the SAE ARP 4754A [‡]. This standard and its impact on this work is analyzed more deeply in a later section.

The runtime monitors proposed in this work are based on a formal temporal specification language, where monitoring properties are formalized using a dedicated language and executable monitors are automatically generated based on them. Generated monitors then contain verified guarantees about resource consumption, e.g., memory, or their execution behavior, e.g., verified computations. There are a variety of temporal languages for specifying monitoring properties [11, 12]. Here, we focus on the stream-based specification language RTLola for which approaches for verified compilations [13] and preliminary work on static analysis of the specification itself exists [14]. In this paper, we extend these efforts of correct safety monitors by examining how objectives of common software development assurance standards such as DO-178 can be achieved automatically. This has been done before for the automatic function of the DLR research platform ARTIS [15] but not in the context of safety monitors.

In [16], the authors utilize a safety monitor approach, where the safety Monitor has a DAL C assurance level and supervises the output of two dissimilar DAL D components. A certification complaints analysis is performed on the DAL C objectives of DO-178C as well as DO-330 objectives. However, the focus is on the machine learning aspects, whereas this work specifically is focused on the monitoring component itself.

II. Applicable Regulations and Standards for UAM

Understanding the regulatory landscape is crucial for successfully implementing runtime assurance in UAM. As this paper deals with runtime monitoring and assurance in the context of autonomous UAM operations ferrying passengers or cargo, it is useful to look at the regulations for manned aircraft as well as unmanned aircraft. This section will also

[†] https://www.sae.org/standards/content/arp6539/

[‡] https://www.sae.org/standards/content/arp4754a/

look at how runtime assurance itself fits into the regulatory framework and how we can leverage the concepts of runtime monitoring to increase the level of automation in UAM operations.

A. Certification Requirements for Manned UAM Operations

Presently in Europe, the European Union Aviation Safety Agency (EASA) has established airworthiness requirements for aircraft intended for UAM operations in the Special Condition for Vertical Takeoff and Landing Aircraft or SC-VTOL. An aircraft in enhanced category of SC-VTOL would have to fulfil safety requirements equivalent to that of manned aircraft in the transport category today, where catastrophic failure conditions leading to a fatal accidents are expected to occur less than once in a billion flight hours. These safety requirements for the development of systems and equipment are then fulfilled by following the guidance provided by industry consensus standards, in turn adopted as acceptable means of compliance (AMCs) by the certification authorities.

The most common standards used in the civil aviation industry today are SAE ARP 4754A (for systems development), SAE ARP 4761 (for safety assessment), RTCA DO-254 (for airborne electronic hardware) and RTCA DO-178C (for software in airborne systems and equipment). A common concept across these standards is that of the development assurance level (DAL), which is derived from safety assessments performed at the aircraft and system levels. Depending on the failure effects of a function, it should be developed to DAL A (Catastrophic), DAL B (Hazardous), DAL C (Major) or DAL D (Minor) respectively. These functional development assurance levels (FDALs) are inherited by the associated item (IDALs) housing the function, with an expectation for the hardware and software components implementing the function to be developed to an equivalent rigor.

At the software level, the standard DO-178C establishes assurance objectives for various aspects of the development lifecycle for the corresponding software levels A-D, such as planning, specification, development and verification. The levels themselves can be further modified by choosing architectural strategies to share the burden of safety among dissmilar redundant functions, which then allows for a lower level of rigor for each redundant component, provided that common cause failures (CCFs) are accounted for in an appropriate manner. In this work, we choose runtime assurance as a novel architectural strategy.

B. Certification Requirements for Automated UAM Operations

While the premise of UAM is based around operations in the urban environmwent, most manufacturers in this domain are proposing autonomous cargo operations in sparsely populated regions due to the operational risk associated with large drones of such novel configurations. Moreover, the UAM market can also include operations in densely populated urban areas using cargo carrying drones of a much smaller weight and size class than those typically associated with the domain. Finally, passenger carrying autonomous UAM aircraft are touted as the harbringers of scalability and therefore profitability to this new market segment. However, these operations are also associated with the highest levels of risk. With this context, we can now look at the airworthiness requirements for unmanned aircraft. The approach to certification of unmanned aircraft is quite different at least in Europe and is explicitly centered around operational risk.

The EU Implementing Regulation (EU) 2019/947 defines three basic categories of drone operations: "open", "specific", and "certified" [17]. Whilst the "open" category covers low-risk operations, which generally require Visual Line Of Sight (VLOS) to steer drones in a controlled environment, the "specific" and "certified" categories are the ones suitable for a larger variety of commercial operations. In the "specific" category, the drone operator needs to obtain an authorization for their individual operation from the national aviation authority based on a risk assessment, which determines the required precautions to safely conduct this operation. This methodology, called the Specific Operations Risk Assessment (SORA) and originally published by an international group of regulatory experts [§] [18], describes both operational and technical means of risk mitigation. Whereas the operational mitigations are enforced through the implementation of robust procedures, the applicant has to show that the technical mitigation means are developed according to industry standards.

As the risk of the operation and the associated SAIL [¶] level increases, the development assurance requirements become more similar to those for manned aircraft. Similar to the SC-VTOL, but much less rigorous in nature and with a maximum takeoff mass (MTOM) of 600 kg, another special condition SC-LUAS defines the initial airworthiness requirements for light unmanned aircraft systems. Applicants following the SORA are required to apply for type

[§]JARUS or the Joint Authorities for Rulemaking on Unmanned Aircraft Systems is an international body of experts on drones from the national aviation authorities, industry and academica who published the original concept of SORA, adopted and further amended by EASA in recent years

[¶]SAIL or specific assurance integrity level is a composite measure of the ground and air risk of a UAS operation in the specific category and ranges from SAIL I to SAIL VI.

certification of their drone on the basis of SC-LUAS for SAIL V and VI operations with a recommendation to use the special condition as a basis for SAIL III and IV operations as well.

For example, in a recently published proposed means of compliance for the safe design of drones for SAIL IV operations, EASA has indicated that functions whose direct failure would lead to the loss of control of the operation need to be assured to DAL C [‖]. Operations with a risk level higher than SAIL VI fall under the certified category and these unmanned aircraft need to be type-certified according to their respective certification bases, e.g. SC-VTOL.

We can therefore see that, for both manned and unmanned aircraft, the common thread to DO-178C is through the corresponding requirements for safety and reliability considerations in the design of the aircraft. While runtime assurance can help us mitigate the DALs of the complex function itself, the monitor must still be developed to the highest assurance level required of the function. In the scope of this paper, we show how runtime assurance can be used to leverage ADS-B data for augmenting active surveillance sensors and enhance the detect-and-avoid performance of the aircraft, while still ensuring the level of safety required from a traditional detect-and-avoid function. We particularly focus on the use of a stream-based monitoring language to specify the runtime monitor validating the ADS-B data and the software certification aspects of this formal specification language itself.

III. Detect-and-Avoid Use Case

Although segregation of UAS operations is a easy way to control ground and air risk of operations, it comes with several constraints that scale with the number of operations (e.g. size of protection areas, flight plan definition). To move beyond these limitations and enable upscaling the number of operations, UAVs need to be integrated in the air traffic. However, this shall not come at the cost of a reduction in safety for the rest of the traffic. For this reason, authorities require integrated UAS to be equipped with a Detect-And-Avoid systems. This system receives input from sensors, processes them to form a representation of the surrounding traffic, uses the representation to alert the pilot/UAS of conflicting encounters, and computes safe maneuvers to avoid conflicts, these maneuvers are proposed to the pilot or directly to the UAV flight controls, depending on the level of automation. Central to the DAA system is the reliability of sensor inputs. The most widespread sensor is the ADS-B/in, it allows sensing the traffic equipped with ADS-B/out (e.g. all commercial aviation).

Though ADS-B is widespread and precise, its integrity is insufficient for a system to propose decisions based on its input alone. Plus, being based on aircraft self-reporting their position through ADS-B/out, it is subject to cyber-security weaknesses (e.g. spoofing). The accepted solution is to "validate" the ADS-B/out signal by confirming it with a second sensor. As stated by a 2017 RTCA white paper [19]: "For DAA systems, the implications of this policy are as follows: DAA Warning guidance may only be generated on ADS-B tracks that have been validated [. . .]. DAA guidance to regain well-clear may only be generated on tracks that have been validated. DAA Caution suggestive guidance may be generated by unvalidated ADS-B tracks. If the DAA system is being integrated with the UAS flight control system to automate maneuvers, the guidance system may only maneuver automatically against ADS-B tracks that have been validated.".

To which extent this sensor input can be used for DAA depends on the stage of the encounter (warning, guidance, caution) and the available information (e.g. for an aircraft not reporting altitude, horizontal maneuvers only will be allowed for avoidance). To guide the avoidance sub-system on the available maneuvers, a monitor is used to validate the ADS-B data and, depending on the result of the validation, allow only certain alerts/maneuvers. The result is a DAA system allows providing safe alerts/maneuvers, with its behavior tailored to the available sensor data.

The requirements to validate an ADS-B/in track are described in RTCA's DO-365 Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems, see Table 1. The requirements are linked to the comparison between the information provided by ADS-B In and an Active surveillance sensor. Though the reasoning behind each requirement is complex, this paragraph gives a notion of the origin of each of the highest-level requirement. Requirement one has been quantified based on a mix of inherited TCAS II, CDTI Assisted Visual Separation (CVAS), In Trail Procedures (ITP) and Aircraft Surveillance Applications (ASA) MOPS requirements. The range thresholds in requirement two come directly from CAVS requirement, though slightly more conservative. Based on these range thresholds, the acceptable bearing error in requirement three has been computed as follows:

$$\texttt{Bearing error} = \textit{arctan}(\texttt{combined position error}/(\texttt{range} - \texttt{combined position error})),$$

where the combined position error is the combination of ownship navigation, intruder navigation, ownship surveillance, latency, and other errors. Using the numbers in requirement two shows that the position can have a maximum allowed

[‖] https://www.easa.europa.eu/en/document-library/product-certification-consultations/means-compliance-light-uas2510

ID	Requirement	Source
1)	Range, Bearing and altitude criteria for ADS-B Position Validation Using active surveillance	DO-365B
a)	$abs(\texttt{slant range difference}) < 0.25NM$	p. N-5
b)	$abs(\texttt{bearing difference}) < 45degrees \wedge \texttt{range} > 1NM$	
c)	$abs(\texttt{altitude difference}) < 200ft$	
d)	TOA between ABS-B position and active surveillance data must be no more than 250ms	
2)	Active surveillance range validation threshold	DO-365B
a)	Ownship horizontal position error (95%) must be no more than $185m$	p. N-6
b)	Traffic horizontal position error (95%) must be no more than $185m$	
c)	Horizontal range error for validation position source (95%, bias $+2\sigma$) must be no more than $69m$	
d)	Uncompensated latency in ownship position ($600kts$ for $250ms$) must be no more than $77m$	
e)	Uncompensated latency in ADS-B traffic ($600kts$ for $600ms$) must be no more than $185m$	
f)	Other data processing errors (including differences in the TOA of position sources) must be no more than $145m$	
g)	Total, assuming errors root sum square must be no more than $367m(.2NM)$	
3)	ADS-B must be validated every ten seconds	DO-365B p. N-10
4)	Active surveillance bearing validation threshold	DO-365B
a)	At slant range greater than $14NM$, bearing error must not be greater than $0.8deg$	p. N-7
b)	At slant range greater than $8NM$, bearing error must not be greater than $1.5deg$	
c)	At slant range greater than $5NM$, bearing error must not be greater than $2.4deg$	
d)	At slant range greater than $2NM$, bearing error must not be greater than $6.3deg$	
e)	At slant range greater than $1NM$, bearing error must not be greater than $14deg$	
f)	At slant range greater than $0.5NM$, bearing error must not be greater than $42deg$	
5)	The UA DAA processor shall receive the [active surveillance] data:	DO-365B
a)	Track ID (note 1)	p. N-46
b)	24-bit Mode S Address (note 2)	
c)	Slant Range	
d)	Relative Bearing (note 4)	
e)	Barometric Pressure Altitude	
f)	Altitude quantization	
g)	TOA betweend ABS-B position and active surveillance data must be no more than 250ms	
h)	Range Valid Flag (note 6)	
i)	Bearing Valid Flag (note 6)	
j)	TCAS equipage (note 7)	
k)	Active/Passive Flag (note 8)	
l)	TCAS Coordination Data (note 9)	

Table 1 ADS-B validation requirements in natural language

error of 367m (0.2 NM). Below 1 NM, the error is too close to the range and the bearing calculation is no longer valid. Requirement number four comes from an assumption that the maximum encounter speed is 600kts (head-on case, with ownship maximum speed of 200kts and traffic going at up to 400kts). This leaves 10 seconds for a revalidation interval

according to the TCAS II MOPS. Requirement number five states that the active surveillance message used as reference to validate ADS-B should be complete.

This section discussed the choice of detect-and-avoid as an example use case to demonstrate the run time assurance framework. In subsequent sections, the system and software aspects of development and certification will be treated in further detail. It will be shown how the runtime assurance of the DO-365 requirements can be implemented using a system architecture per the guidelines of the ASTM F3269 standard practice. Particular focus is given to the use of RTLola in the formal specification of the runtime monitor to validate the ADS-B data and the software aspects of certification are discussed in detail.

IV. System Level Considerations

A. Review of RTA Architecture from ASTM F3269

A simplified version of the RTA ASTM F3269-21 architecture is shown in Figure 1. Note that in its full version, data is separated into assured and unassured data and data preparation components are added. The runtime assured component is a *complex function* which can be, e.g., a non-deterministic control or a machine-learned function. The safety layer is represented by a *safety monitor* that receives the external data used and outputs produced by the complex function. If the safety monitor detects a violation of its expected behavior, the *switch* mitigates the effects by changing to a *recovery control function*. As a result, the overall system behavior is bounded within safe limits.

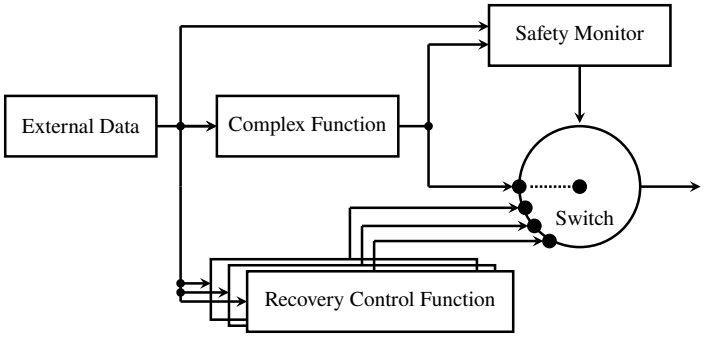


Fig. 1 Runtime assurance architecture proposed by ASTM F3269-21 to safely bound a complex function using a safety monitor.

B. Detect-and-Avoid Use Case in an RTA Architecture

Figure 2 depicts a proposed architecture for the detect-and-avoid use case in the style of a Run-Time Assurance (RTA) architecture as described in the industry consensus standard ASTM F3269. Herein, the Traffic Alerting and Resolution Function, intended to process sensor inputs about cooperative traffic and create alerts and resolution advisories as well as perform collision avoidance if needed, represents the Complex Function. This function is considered "complex" because the outputs produced by this function are based on unassured inputs, viz. ADS-B data and cannot be used directly for traffic alerting or resolution as per the requirements of RTCA DO-365.

Furthermore, the function in this case is also assumed as not being designed to any aerospace standards or requirements, i.e. it is a black-box whose output is untrusted. In this case, an alert or resolution action would first need to be validated by a Safety Monitor to ensure that only trustworthy outputs are given to the pilot or flight guidance computer. In this use case, the "Validation of ADS-B data" block represents the Safety Monitor in ASTM F3269. The "Validity checking of ADS-B data" takes ADS-B and active surveillance sensor data as inputs.

It then feeds the Safety Monitor with feasible ADS-B tracks after conditioning the data (e.g. type correctness, sorting in a suitable format) and checking them for dynamic consistency (e.g. values are within bounds, data is fresh and change trends are reasonable). The Safety Monitor after validating the data provided by the ADS-B and active surveillance sensors, decides the scope of the Traffic Alerting and Resolution function outputs that can be safely forwarded to the pilot or flight guidance computer. It passes on this recommendation to the Scope Selector, which represents the RTA Switch in F3269. The Scope Selector interfaces to the Complex Function should be designed in such a way to allow for this modular scope selection and output forwarding.

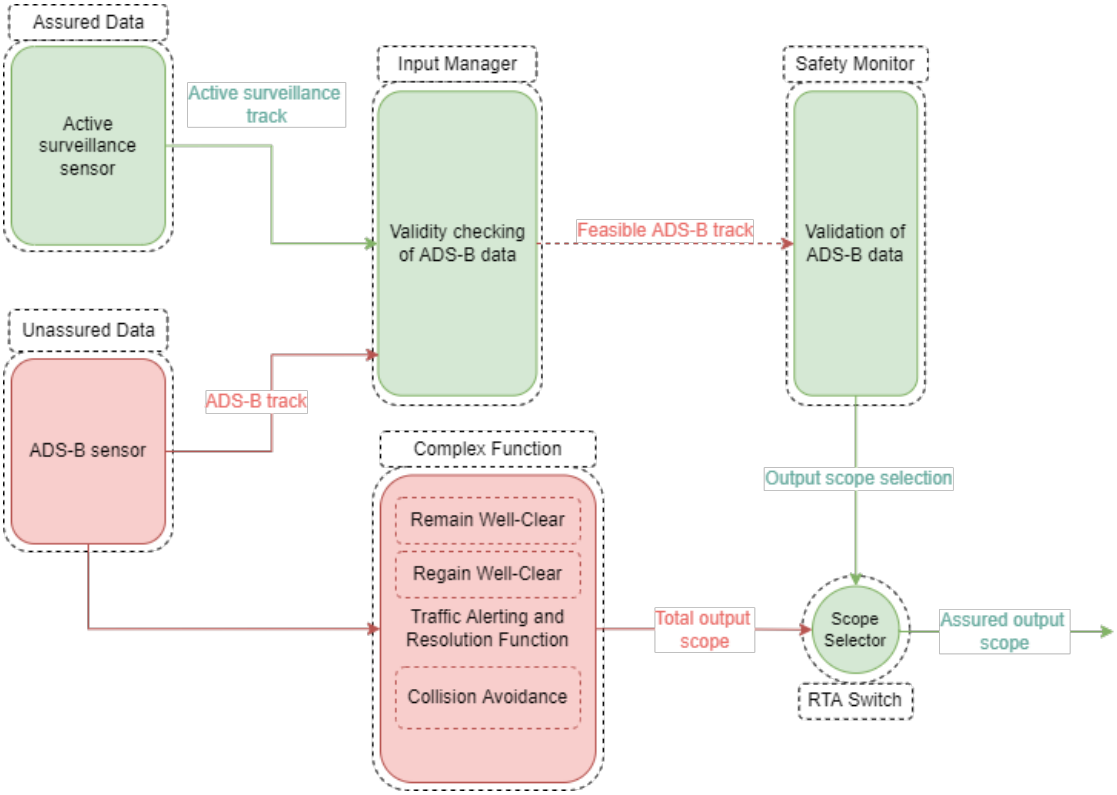


Fig. 2 Architecture for a detect-and-avoid use case superimposed over the ASTM F3269 standard RTA architecture

Notable in Figure 2 is the absence of a Recovery Function. Since in this work, the detect-and-avoid function is used to augment the pilot's situational awareness and enhance operational safety in visual meteorological conditions (VMC), a form of recovery function would simply be to passivate the complex function. To extend this further, the safety monitor could allow a reduced scope of the complex function output which it can guarantee to be assured. This way, the complex function itself in various subsets of reduced scope acts as a recovery function. The available reduced scopes are: no warning, no caution (limiting the alert leves); vertical only, horizontal only (limiting manoeuvers). This way, maximum performance of the function is allowed and still safely bound the output of the complex function. We believe that this modified RTA architecture could represent all complex functions where fail-safe behavior is an acceptable outcome. Of course, this architecture would only apply for modular complex functions where reduced or degraded performance is still more useful than to have than no function.

This section showed a representation of the DAA use case architecture in the form of a standard RTA architecture as per ASTM F3269. It also provided a brief discussion of deviations from the standard RTA architecture and depicted a modified architecture to represent the bounding of fail-safe complex functions without the need for separate recovery functions. As discussed in section III the requirements to validate an ADS-B/in track are described in RTCA's DO-365 Minimum Operational Performance Standards (MOPS), see Table 1. For the purpose of runtime assurance, these requirements have been transcribed into specifications and written in the RTLola language (see Figure 6).

V. Monitoring with RTLola for Runtime Assurance

Unlike traditional static analysis methods that focus on detecting errors before execution, runtime assurance operates during execution, monitoring and verifying behaviors of a system as it runs and activating counter-measures in case of system violations. In this context, a violation is a behavior that does not match the monitoring specification. And the behavior is a combination of observable data stream sequences. As an important objective for certification of software the seamless traceability from system requirements to the executable object code is identified in section VI. Given the context of this complete traceability, the idea is that system requirements and system safety requirements are developed

into executable monitors. Any violation of a monitor can then be traced back to a violation of system behavior. By identifying these violations during operation it is possible to trigger mitigations and therefore keep the operation safe.

RTLola is a stream-based monitoring framework for cyber-physical systems. Given a specification in the RTLola specification language, the framework generates a runtime monitor observing the system during its execution. If a violation is detected, the monitor notifies the user so a counter-action can be performed to correct the system's behavior. RTLola specifications are statically analyzed to guarantee a safe execution at runtime. These analyses include the existence of a unique and safe evaluation, the required memory, and the order in which the streams are evaluated. With this information, the RTLola framework either directly executes the specification with the RTLola interpreter [20], synthesizes the specification onto an FPGA [21], or compile the specification to C or Rust code.

Section V.A introduces the RTLola specification language. We describe the general concept based on two requirements from Table 1. Section V.A.1 provides an overview of the compiled specification as Rust code. This code is an executable realization of the specification with the same semantics. This paper uses a compilation to provide a certification path for RTLola Monitors as discussed in Section VI.

A. RTLola Specification Language

In RTLola, we differentiate between three types of streams. First, input streams represent incoming data that arrive as timed and possibly asynchronous data streams. Second, stream equations translate these data streams into output streams to process, evaluate, and aggregate the input data. They play a similar role as variable assignments in standard programming languages by filtering incoming data, comparing values from different streams, or carrying out more complex computations. Third, trigger streams contain boolean expressions describing a specific noteworthy boolean evaluation, for example, a transition to an unsafe state. Compared to standard programming languages, RTLola provides temporal operators to specify temporal requirements. These operators either refer to past stream values or perform computations as aggregating past values.

Consider the specification in Figure 3 as an example of an RTLola specification. Here, we describe requirements 3 and 4 of Table 1. A specification typically starts with the declaration of the data given to the generated monitor. These declarations, called input streams, consist of a name and a type. The name of a stream acts as an identifier used in the stream expression to access the stream values. The type of stream describes the bit size and its interpretation. More concretely, the input stream `ads-b_time_of_validity`, for example, contains the timestamp of the ADS-B validity check (in seconds) as a float value of 64 bits.

Requirement 3 is a frequency check that counts the number of received data points and compares this count with a threshold. We use the `time_of_validity` stream to represent the frequency of the ADS-B and declare the output stream `requirement_3` to describe this check. This output stream uses the `aggregate`-operator, which takes as arguments the stream over which the values are aggregated, the duration of the aggregation window, and the aggregation function describing the computation. More concretely, we use the count aggregation function counting the number of `time_of_validity` value for a duration of 10 seconds. If this value is smaller than 1, the system does not receive the ADS-B values with the required frequency resulting in an unsafe state of the system. This unsafe state is described with the following trigger declaration. If `requirement_3` is violated at some point, the trigger activates and the corresponding message is provided to the user.

Requirement 4 compares the `bearing_error` against a threshold that depends on the `slant_range`. We use the dynamic filtering technique of RTLola to describe this dependent check. The different eval-clauses consist of the `when` and the `with` expression. The expression after the `when`-keyword describes the dynamic filter — in this example, the `slant_range` check — the expression after the `with`- keyword describes the stream value – in this example, the bearing error checks. As a result, we compare the `bearing_error` dependent on the `slant_range` as intended. The stream equation in `slant_range` stream computes the Euclidean distance between the position computed by the ADS-B and the vehicle's position. Since the positions are received as latitudes and longitudes values in radian, and the computation uses ECEF coordinates, the streams `ads_b_coord` and `own_coord` converts these positions. These computations only depend on the current value and use mathematical and trigonometric functions, so we outsource these computations for readability reasons. The `bearing_error` stream follows its description, which is similar to standard programming languages.

Based on the stream declarations, the RTLola framework analyses the dependencies between streams. The RTLola framework visualizes this information in the so-called *Dependency Graph*. In this graph, the nodes are labeled with the stream names, and the edges are marked with the stream accesses in the stream declaration. Additionally, the RTLola framework provides static guarantees about the runtime behavior using this graph. These guarantees include 1) the existence of a unique evaluation, 2) the memory bound per stream describing the number of past stream values used in

```
input ads-b_time_of_validity : Float64 /// in seconds
input ads_b_lat_rad : Float64
input ads_b_lon_rad : Float64
input ads_b_alt_ft: Float64
input own_lat_rad : Float64
input own_lon_rad : Float64
input own_alt_ft : Float64

/// Requirement 3
output requirement_3 @10s := b_time_of_validity.aggregate(over: 10s, using: count) >= 1
trigger ¬ requirement_3 "Invalid frequency for ADS-B"

/// Requirement 4
output abs_b_coord := wgs84_to_ecef(ads_b_lat_rad, ads_b_lon_rad, ads_b_alt_ft)
output own_coord := wgs84_to_ecef(own_lat_rad, own_lon_rad, own_alt_ft)
/// Euclidean distance between airship and ads_b
output slant_range_m := sqrt((abs_b_coord.0 - own_coord.0)^2 + (abs_b_coord.1 - own_coord.1)^2 + (abs_b_coord.2 -
     own_coord.2)^2)
constant COMBINED_POS_ERROR_M := 370.4
output bearing_error := arctan(COMBINED_POS_ERROR_M / (slant_range_m - COMBINED_POS_ERROR_M))

output requirement_4
   /// Requirement 4a
   eval when slant_range > 14.0 with bearing_error > 0.8
   /// Requirement 4b
   eval when slant_range > 8.0 with bearing_error > 1.5
   ...
trigger ¬ requirement_4 "Active surveillance bearing error threshold violation"
```

Fig. 3 RTLola specification for the Properties 3 & 4 in Table 1

the stream declarations, 3) the order in which the streams in the specification are evaluated.

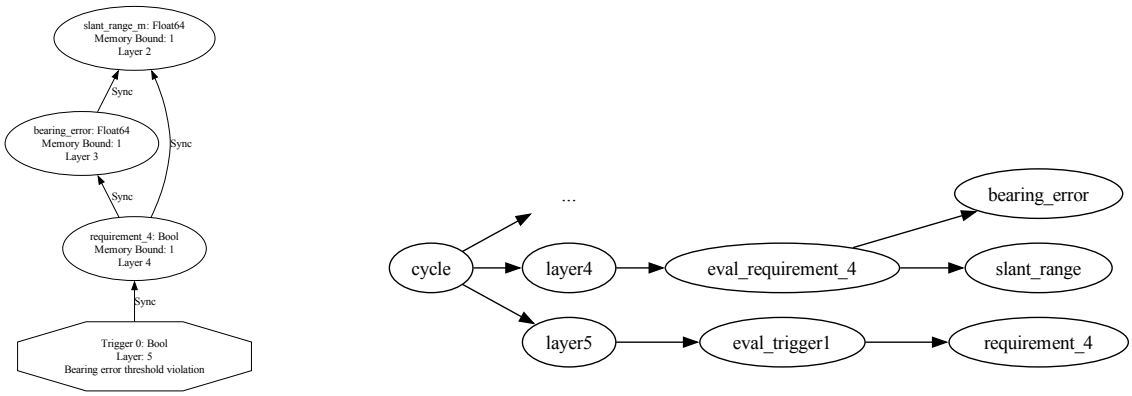
Figure 4a visualizes a dependency graph for a subset of our example. For simplicity, we only visualize the stream `requirement_4` and its ingoing and outgoing dependencies. Besides the visualization of the dependencies, the graph also shows the results of the previously mentioned analyses. More concretely, first, the analysis checked the existence of a unique evaluation model. Such a model exists if there are no cyclic dependencies in the specification that all refer to the current value of a stream. Our subset does not contain a cycle, so a unique evaluation model for this subset exists. Next, the dependency graph contains a memory bound for each stream. This bound determines the number of stream values needed for the correct execution of the monitor. The specification only accesses current stream values, so the memory bound for each stream this stream is 1. In RTLola, the order of the stream declarations does not influence their execution but the dependencies between streams. We need to guarantee that if a stream value is used in a stream equation, this value is already available, i.e., this stream equation was evaluated before. For this, the order of the stream execution is represented by assigning each stream to an evaluation layer. In these layers all streams only access stream values from past evaluations or from previous layers implementing a partial order of the streams. More concretely, the analysis assigned the `requirement_4` stream to layer 4. We can see that the incoming edge starts from the trigger node that has a higher layer, describing a later evaluation, and the outgoing edges point to nodes with a smaller layer.

1. Realization

In this paper, we compile RTLola specifications to Rust code. Additionally, the compiler provides the user with a fully annotated specification, a graphical representation of the analyzed information of the specification, and the call graph of the compiled code. We extended the compilation with annotations to relate the compiled code with the RTLola specification. These annotations include: 1) a reference to the caller and callee of the function to relate the generated code with the generated call graph 2) a mapping to the semantical part of the specification, a function or field is implementing 3) an explanation of the required memory and the control flow of the program.

In general, the compiled code is separated into six parts: 1) an external interface starting the evaluation of a monitor cycle 2) a schedule that decides for each timestamp if a deadline of a periodic stream is reached 3) a function `cycle` implementing an evaluation cycle by iterating over each layer 4) `layer`-functions iterating over the streams in this layer and calling their evaluation function if activated 5) the evaluation function that evaluates the stream equation and updates the stream memory 6) a memory layout containing the stream values and providing getter and setter functions to access these values .

Figure 5 presents a short example for each part to give an intuition of the compiled code. Only code excerpts are



(a) Partial Dependency Graph **(b) Partial Call Graph**

Fig. 4 Visualizations of specification and its realization

shown, but the principle is transferable to the complete specification. For readability, we use … inside the functions if the code follows the same structure.

The monitor compiles the `accept_event` function as an external interface to accept incoming data and to execute the monitor with this input and timestamp. This function translates the input to the internal representation presented in Figure 5a. This representation has an optional field for each input stream, indicating that this stream receives new data. The type of these fields corresponds to the type declared in the specification. We annotate each field with the input stream declaration in the specification to validate the data type. Additionally, the internal representation contains a boolean flag for each period in the specification. This flag is activated from the static schedule, activating streams annotated with this frequency. We annotate these fields with the corresponding output streams to describe this relation. After the evaluation, the function returns all activated trigger encoded in the `Verdicts` struct.

Figure 5b implements this static schedule for all periodic streams in the specification. This functionality is in the RTLola specification reflected in streams annotated with a frequency. These streams are activated at fixed points corresponding to the source code's deadlines. We initialize the schedule with the `new` function by collecting all deadlines and annotate this function with all periodic streams to validate the completeness of the schedule. Additionally, each state in the schedule is annotated with each stream activated by the deadline. In our example, we only have a deadline of 10 s, resulting in a schedule with one state activated every ten seconds. The `next` function is called by the `accept_event` function and implements the schedule. It decides if a deadline is reached between the current timestamp given with the `end` argument and the last timestamp that is stored in the `Queue`. If a deadline is reached, the function returns an internal state, activating all streams within this period. In our example, this condition is true every ten seconds encoded in the single state `state0`. Since this function is independent of the specification, we do not need to annotate it with concrete information in the specification.

Figure 5c represents the `cycle`-function from our example. This function takes as input the internal state and iterates over all evaluation layers. For the documentation and to validate that every stream is covered by precisely one layer, we annotate the single-layer calls with the stream in this layer. In our example, only the output stream `requirement_4` is evaluated in layer 4, so we annotate this stream as documentation. As a last step, we build the verdict from the output streams if all streams are evaluated. For this, the function takes the current value of all triggers in the specification and stores this value in the `Verdict` struct.

Figure 5d presents a code snippet of the `layer_1` function. This function iterates over all streams in layer 1 and calls their evaluation function. We annotate these functions at the beginning with all streams inside this layer. With this approach, we can validate that every `eval` function is called and can compare the annotated streams with the information in the generated dependency graph and call graph. To decide if a stream is evaluated with the evaluation function call, we need to check if this stream is activated. This activation can depend on the control flow of the stream or a deadline. In our example, the stream `own_coord` that converts the vehicle's latitude, longitude, and altitude from radian and feet to meters is only evaluated if the monitor gets a new latitude, longitude, and altitude value. This condition is reflected in the specification with the `@` identifier, so we annotate this code fragment with this information. In comparison, the stream

```rust
struct Internal {
    // input ads_b_time_of_validity : Float64
    pub ads_b_time_of_validity: Option<f64>,
    ...
    // output requirement_3 @10s
    static10000: bool,
    time: Duration,
}
```

(a) Internal Reprentation

```rust
impl Monitor {
    fn cycle(&mut self, event: Internal) ->
         Result<Verdict, MonitorError> {
        ...
        // Layer 4: requirement_4
        self.layer4(&event)?;
        ...
        Verdict::new(self)
    }
}
```

(b) Schedule

```rust
impl Queue {
    // Create a new schedule
    // Periodic streams:
    // output requirement_3 @10s
    fn new(start_time: Duration) -> Self {
        // output requirement_3 @10s
        let state0 = State {
            duration: Duration::new(10, 0)
            time: start_time + Duration::new(10, 0),
            deadline: Deadline::Static10000,
        };
        Self(Queue::from(vec![state0]))
    }

    // Returns the next deadline that is reached
    fn next(&mut self, end: Duration) -> Option<Internal> {
        ...
    }
}
```

(c) Layer Iteration

```rust
// Layer 1: abs_b_coord, own_coord, requirement_3
fn layer1(&mut self, event: &Internal) -> Result<(),
     MonitorError> {
    ...
    // output own_coord @own_alt_ft
    if event.own_lat_rad.is_some() &&
         event.own_lon_rad.is_some() &&
         event.own_alt_ft.is_some() {
        self.eval_own_alt_m()?;
    }

    // output requirement_3 @0.1Hz
    if event.static10000 {
        self.eval_requirement_3()?;
    }
    Ok(())
}
```

(d) Stream Layer

```rust
// RtLola definition:
// output bearing_error: Float64
// eval with arctan(370.4 / (slant_range_m - 370.4))
fn eval_bearing_error(&mut self) -> Result<(),
     MonitorError> {
    let new_value = 370.4f64 / self.slant_range_m()? -
         370.4f64.atan();
    self.stream_memory.bearing_error.update(new_value)?;
    Ok(())
}
```

(e) Stream Evaluation

```rust
struct StreamMemory {
    ...
    // Memory for stream "slant_range_m" of type Float64.
    // Memory bound 1 because it is accessed by
         "bearing_error" with Sync.
    // All accesses to this stream:
    // - "bearing_error" with [Sync].
    // - "requirement_4" with [Sync].
    slant_range_m: StreamBuffer<f64, 1>,
    ...
}
```

(f) Memory Layout

Fig. 5 Examples of the compiled code for the specification in Figure 3

`requirement_3` is evaluated every ten seconds. This condition is in the internal representation with the `static10000`-flag resulting from the schedule in Figure 5b.

Figure 5e presents the evaluation of the `bearing_error` stream. First, this function computes the new stream value following the stream declaration, as annotated in the example. During the computation, the code calls the corresponding getter functions for the different stream lookups. We can validate these lookups by comparing the outgoing dependencies in the dependency graph with those in the call graph. Then, the memory is updated with the computed value.

Figure 5f gives an overview of the memory layout of the compiled code. For each stream in the specification, this struct contains a field type with the generic `StreamBuffer` type. This type takes as arguments the type of the stream values and the memory bound. We annotate each stream with the type inferred by the RTLola specification, the computed memory bound, and the dependencies to validate these arguments. The `Streambuffer` type itself is a generic ring-buffer that is independent of the concrete specification.

Figure 4b partially presents the call graph of the program. The call graph visualizes the program's control flow, allowing the reviewer to connect the code's control flow with the control flow of the specification encoded in the dependency graph. The segment starts with the `cycle` function iterating over the `layer` functions. The `layer` functions, then call the evaluation of the streams using lookup functions to access other stream values. First, we identify that the annotated layer in the dependency graph corresponds with the function call of the layer function in the call graph

exemplified with the `requirement_4` stream. Additionally, we observe that outgoing dependencies in the call graph correspond to lookup functions in the call graph. For example, the `eval_requirement_4` function calls the `slant_range` and `bearing_error` lookup function. The same observation holds for the outgoing edges in which the `eval` function calls the lookup functions of the stream. In our example, the `eval_trigger1` function calls the `requirement_4` function to access the current stream value.

VI. Software Aspects of Certification

Different strategies can be used for the certification of a runtime monitoring approach. These range from manually reviewing and certifying the generated C code to qualifying the RTLola generator as a tool. Using a supplement for model-based development as well as for formal methods also offers potential benefits for certification. This section briefly outlines the two approaches for safe monitoring software: certification based on tool qualification DO-330 in Subsection VI.A and certification based on DO-178C in Subsection VI.B. It then outlines one approach in more detail, discussing certification objectives, and the compliance rationale that can be used when utilizing our approach using RTLola runtime monitoring.

A. Certification Using DO-330 Tool Qualification

This approach utilizes the DO-178C standard in combination with DO-330 standard for Tool Qualification. In this approach, the compilation from the formal RTLola specification to source code could be qualified as a tool. Tool qualification is differentiated into five levels: TQL-1 to TQL-5. The corresponding TQL depends on the software level and one of three criteria that the tool matches. Since the tool is used for generating code, this would match the criteria 1: "A tool whose output is part of the airborne software and thus could insert an error". This approach has the potential to reduce the overall certification effort. However, in this work we focus on the traditional certification (see below), leaving the detailed discussion of this approach for future work.

B. Certification Using DO-178C

This is the basic approach that utilizes the DO-178C standard alone. We focus on this approach, since it is the basis to certification. Additional supplements are still based on the underlying DO-178C standard and give the possibility to eliminate or exchange specific objectives, given that additional objectives of the supplement are met. With this section, we want to show that the monitoring is certifiable and can thus be used in an airtaxi, or larger cargo drone, a system that requires certification.

With this approach, the requirements, design and code is verified traditionally, by reviews and tests. There is no specific credit applied for the monitoring approach or the generated code. Nonetheless, it is interesting to discuss how the certification objectives can be fulfilled by utilizing the RTLola runtime monitoring approach. The objectives will be discussed in more detail in the following sections.

C. Discussion of Objectives

In the following we discuss the objectives for Subsection VI.B. In our approach, we define High-level requirements as requirement given in natural language similar to Table 1 and low-level requirements correspond to RTLola specifications. The complete table can be viewed in the appendix Table 2.

Table A-1.5 Software development Standards This objective asks for the establishment of standards for software requirements, software design, and software code. With the use of RTLola, the high-level software requirements are translated to the formal language. Since RTLola is a formal language, the standard is established automatically utilizing the language and its semantic. More concrete, if the user provides a specification that does not satisfy the standard for example by having an unsafe state, the RTLola framework does not generate any source code. Furthermore, the software code is generated by our compiler. For this, a standard can be defined.

Table A-2.1 High-level requirements High-level requirements are developed traditionally in natural language. This is the starting point for defining RTLola requirements. Requirements get a unique ID that will be traced throughout the further development and verification processes. The specification in fig. 3 exemplifies this with the comments.

Objectives Table	Description	Applicable level	Compliance rationale for RTLola runtime monitoring
A-1.5	Software development standards are defined	A-E	(Semi-)Established by use of formal language.
A-2.1	High-level requirements are developed	A-D	Traditional natural language requirement.
A-2.4	Low-level requirements are developed	A-C	Maps to the formal RTLola language specification.
A-2.6	Source code is developed	A-C	Source code is generated by the RTLola compiler, however code is human readable and can be certified traditionally.
A-4.2	Low-level requirements are accurate and consistent	A-C	Formal semantics ensures unambiguity.
A-4.4	Low-level requirements are verifiable	A-B	Compilation of requirements ensures verifiability.
A-4.5	Low-level requirements conform to standards	A-C	Formal semantics ensures conformity.
A-4.6	Low-level requirements are traceable to high-level requirements	A-C	Use of unique ID.
A-5.1	Source code complies with low-level requirements	A-C	Automated compilation ensures compliance.
A-5.3	Source code is verifiable	A-B	Automated compilation ensures compliance.
A-5.4	Source code conforms to standards	A-C	Automated compilation ensures compliance.
A-5.5	Source code is traceable to low-level requirements	A-C	Automated compilation ensures compliance.
A-5.6	Source code is accurate and consistent	A-C	Automated compilation ensures compliance.
A-6.3	Executable object code complies with low-level requirements	A-C	Automated compilation ensures compliance.
A-8.2	Baselines and traceability are established	A-D	Formal language enables traditional use.
A-8.6	Software life cycle environment control is established	A-D	RTLola language and compiler language to be archived.

Table 2 Certification aspects of runtime monitoring

Table A-2.4 Low-level requirements The low-level requirement is developed from the high-level requirement as formal RTLola specification. For traceability purposes the unique ID from the high-level requirement is included before the formula as a comment.

Table A-2.6 Source code Code is generated by the RTLola compiler from the formal specification. The unique requirement ID is automatically transcribed into the respective source code blocks that are generated for this requirement. Note that in our approach we generate the source code out of the low-level requirements.

Table A-4.2 Low-level requirements are accurate For this objective it is necessary to show that each low-level requirement is accurate and unambiguous. Since the requirements are formulated in a formal language, this is the case. The semantic of RTLola is deterministic and so is the low-level requirement. Furthermore, low-level requirements should not conflict with each other. While this is a problem for traditional development, for stream monitoring this is not an issue. The compiler makes sure that all streams can be evaluated: For example, a cycle-check guarantees a cycle-free evaluation, so there are no recursive calls between stream evaluations.

Table A-4.4 Low-level requirements are verifiable In addition to the rational above (low-level requirements are accurate), the requirement is verifiable if the compiler successfully generates the source code.

Table A-4.5 Low-level requirements conform to standards The low-level requirements are formulated in the formal language and therefore automatically conform to the formal syntax. This is automatically checked by the RTLola compiler.

Table A-4.6 Low-level requirements are traceable This is ensured by using the unique ID provided by the high-level requirement. See the example for details on the traceability.

Table A-5.1 Source code complies with low level requirements This is ensured by the automated compilation of the formal specification to source code. However, the source code is human readable and can be read by reviewers.

Table A-5.3 Source code is verifiable Similar to the last objective, this is ensured by the automated compilation of the formal specification to source code. However, the source code is human readable and can be read by reviewers.

Table A-5.4 Source code conforms to standards The source code is generated. The code can be designed to conform to a coding standard.

Table A-5.5 Source code is traceable As already mentioned in Objective A-2.6, the source code is generated from the RTLola compiler from the formal specification. For traceability, the unique requirement ID is automatically transcribed into the respective source code blocks that are generated for this requirement. Additionally, we extended the compilation with stream-based specific annotations to relate the source code with the specification. We exemplified in section V.A.1.

Table A-5.6 Source code is accurate The DO-178C standard requires accuracy and consistency of the source code. This means that e.g., memory usage, floating-point arithmetic, worst-case execution timing, and other the software aspects have to be shown to be valid and adequate for the system. RTLola specifications are analyzed statically to guarantee a safe execution at runtime. This analysis includes a memory computation to determine the head and stack memory of the specification. Since, we compile the code from the specification, we avoid overflows using check arithmetic operators.

Table A-6.3 Executable object code complies with low-level requirements This is ensured by the automated compilation of the formal specification to source code. However, tests can be executed to check this. Further, note that certified compilers exist, most recently also for Rust**.

Table A-8.2 Baselines and traceability established The RTLola specification is text-based and can be handled similar to source code. Baselines and traceability can therefore be established traditionally.

Table A-8.6 Software lifecycle Regarding the software lifecycle it must be assured that the version of RTLola compiler is frozen with start of the development. Furthermore, complete source code, documentation and binary of the compiler must be archived. However, RTLola specifications are structured into multiple streams, which can be modified and reused with a high degree of flexibility. This allows an easier adaption of the specification during the development. Since the compiler follows strict pattern, the changes in the specification are reflected in the generated code as well and a human review can validate the changes.

**https://ferrous-systems.com

VII. Conclusion

This paper discusses runtime assurance for UAM from a certification perspective. As a first step, the regulatory framework was described. Then, a break down to system and software standards was detailed. As a use case DAA was utilized from the standard MOPS requirements. The DAA architecture was compared to the RTCA runtime assurance standards architecture. Furthermore, the requirements have been formalized to RTLola requirements where RTLola is a formal specification language that allows to automatically generated safety monitors. As an example, we analyzed certification objectives on the software level, e.g., traceability, and discussed how the compliance may be reached by utilizing RTLola. Traceability is of specific interest regarding the use of the RTLola specification language. On the one hand, the specification language replaces a level of specification. On the other hand, monitoring code is generated in a human readable manner, which requires rigorous traceability and thus reduces the gap between natural language requirement and implementation. The traceability across the specification levels is therefore required to support the trustworthiness and verification of this approach. As a result, integrating RTLola and runtime assurance into UAM is a significant step towards enhanced safety and reliability. By considering the regulatory, system, and software aspects, we hope to support the utilization of runtime assurance in UAM systems. Future work will further investigate certification considerations for the supplements to the DO-178C certification standard, such as tool qualification.

Acknowledgment

This work was supported by the Aviation Research Program LuFo of the German Federal Ministry for Economic Affairs and Energy as part of "Volocopter Sicherheits-Technologie zur robusten eVTOL Flugzustandsabsicherung durch formales Monitoring" (No. 20Q1963C).

References

[1] 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 32*, Springer, 2020, pp. 28–39.

[2] Peterson, E. M., DeVore, M., Cooper, J., Carr, G., DeVore, M., Cooper, J., and Carr, G., "Run Time Assurance as an Alternate Concept to Contemporary Development Assurance Processes," , Jan 2020. URL https://ntrs.nasa.gov/citations/20200003114, [Online; accessed 16. Jun. 2022].

[3] DeVore, M., Cooper, J., Wallington, A., Crouse, R., Tsikalas, G., Verma, K., Fleming, C. H., Carr, G., Kirby, N., Cooper, J., Wallington, A., Crouse, R., Tsikalas, G., Verma, K., Fleming, C. H., Carr, G., and Kirby, N., "Run Time Assurance for Electric Vertical Takeoff and Landing Aircraft," , Jan 2022. URL https://ntrs.nasa.gov/citations/20210026909, [Online; accessed 16. Jun. 2022].

[4] Seto, D., Krogh, B., Sha, L., and Chutinan, A., "The simplex architecture for safe online control system upgrades," *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*, Vol. 6, IEEE, 1998, pp. 3504–3508.

[5] Cofer, D., Amundson, I., Sattigeri, R., Passi, A., Boggs, C., Smith, E., Gilham, L., Byun, T., and Rayadurgam, S., "Run-Time Assurance for Learning-Based Aircraft Taxiing," *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–9. https://doi.org/10.1109/DASC50938.2020.9256581.

[6] Schirmer, S., and Torens, C., "Safe Operation Monitoring for Specific Category Unmanned Aircraft," *Automated Low-Altitude Air Delivery - Towards Autonomous Cargo Transportation with Drones*, Springer, 2021, pp. 393–419. URL https://elib.dlr.de/145080/.

[7] Desai, A., Ghosh, S., Seshia, S. A., Shankar, N., and Tiwari, A., "SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems," *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 138–150. https://doi.org/10.1109/DSN.2019.00027.

[8] Wang, Q., Kou, G., Chen, L., He, Y., Cao, W., and Pu, G., "Runtime assurance of learning-based lane changing control for autonomous driving vehicles," *Journal of Circuits, Systems and Computers*, Vol. 31, No. 14, 2022, p. 2250249.

[9] ASTM International, "Standard Practice for Methods to Safely Bound Behavior of Aircraft Systems Containing Complex Functions Using Run-Time Assurance," 2021. URL https://www.astm.org/f3269-21.html.

[10] 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., VIRTUAL EVENT, 2021, pp. 1–11. URL https://arc.aiaa.org/doi/10.2514/6.2021-0525.

[11] Deshmukh, J. V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., and Seshia, S. A., "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, Vol. 51, 2017, pp. 5–30.

[12] Schumann, J., Moosbrugger, P., and Rozier, K. Y., "R2U2: monitoring and diagnosis of security threats for unmanned aerial systems," *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, Springer, 2015, pp. 233–249.

[13] Finkbeiner, B., Oswald, S., Passing, N., and Schwenger, M., "Verified rust monitors for lola specifications," *International Conference on Runtime Verification*, Springer, 2020, pp. 431–450.

[14] Dauer, J. C., Finkbeiner, B., and Schirmer, S., "Monitoring with verified guarantees," *International Conference on Runtime Verification*, Springer, 2021, pp. 62–80.

[15] Torens, C., Adolf, F.-M., and Goormann, L., "Certification and Software Verification Considerations for Autonomous Unmanned Aircraft," *Journal of Aerospace Information Systems*, Vol. 11, No. 10, 2014, pp. 649–664. URL http://dx.doi.org/10.2514/1.I010163.

[16] Dmitriev, K., Schumann, J., Bostanov, I., Abdelhamid, M., and Holzapfel, F., "Runway Sign Classifier: A DAL C Certifiable Machine Learning System," *2023 IEEE/AIAA 42st Digital Avionics Systems Conference (DASC)*, IEEE, 2023.

[17] European Union, "COMMISSION IMPLEMENTING REGULATION (EU) 2019/947 of 24 May 2019 on the rules and procedures for the operation of unmanned aircraft," , 2019-06-11.

[18] Joint Authorities of Rulemaking of Unmanned Systems, "JARUS guidelines on Specific Operations Risk Assessment (SORA)," , 2019.

[19] Radio Technical Commission for Aeronautics, SC-228, "White Paper: FAA position on use of ADS-B for Alerting and Guidance," *RTCA*, 2017.

[20] 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 - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, Lecture Notes in Computer Science, Vol. 11561, edited by I. Dillig and S. Tasiran, Springer, 2019, pp. 421–431. https://doi.org/10.1007/978-3-030-25540-4_24, URL https://doi.org/10.1007/978-3-030-25540-4_24.

[21] 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, pp. 88:1–88:24. https://doi.org/10.1145/3358220, URL https://doi.org/10.1145/3358220.

Appendix

```
import math

constant EARTH_SEMIMAJOR_AXIS : Float64 :=  6378137.0
constant EARTH_FIRST_ECCENTRICITY_SQ : Float64 := 0.00669437999014
constant METERS2FEET : Float64 := 3.280839895

input ads_b_time_of_validity : Float64 /// in seconds
input ads_b_lat_rad : Float64
input ads_b_lon_rad : Float64
input ads_b_alt_ft: Float64
input own_lat_rad : Float64
input own_lon_rad : Float64
input own_alt_ft : Float64

/// Requirement 3
output requirement_3 @10s := ads_b_time_of_validity.aggregate(over: 10s, using: count) >= 1
trigger ¬ requirement_3 "Invalid frequency for ADS-B"

/// Helper Streams

/// Convert ADS-B Lat / Lon / Alt
output ads_b_alt_m := ads_b_alt_ft / METERS2FEET
output ads_b_n := EARTH_SEMIMAJOR_AXIS / sqrt(1 - (EARTH_FIRST_ECCENTRICITY_SQ * (sin(lat) ** 2)))
output ads_b_x := (ads_b_n + ads_b_alt_m) * cos(ads_b_lat_rad) * cos(ads_b_lon_rad)
output ads_b_y := (ads_b_n + ads_b_alt_m) * cos(ads_b_lat_rad) * sin(ads_b_lon_rad)
output ads_b_z := (ads_b_n * (1 - EARTH_FIRST_ECCENTRICITY_SQ)) * sin(ads_b_lat_rad)
/// Convert own Lat / Lon / Alt
output own_alt_m := own_alt_ft / METERS2FEET
output own_n := EARTH_SEMIMAJOR_AXIS / sqrt(1 - (EARTH_FIRST_ECCENTRICITY_SQ * (sin(lat) ** 2)))
output own_x := (own_n + own_alt_m) * cos(own_lat_rad) * cos(own_lon_rad)
output own_y := (own_n + own_alt_m) * cos(own_lat_rad) * sin(own_lon_rad)
output own_z := (own_n * (1 - EARTH_FIRST_ECCENTRICITY_SQ)) * sin(own_lat_rad)
/// Euclidean distance between airship and ads_b
output slant_range_m := sqrt((ads_b_x - own_x) ** 2 + (ads_b_y - own_y) ** 2 + (ads_b_z - own_z) ** 2)
constant COMBINED_POS_ERROR_M := 370.4
output bearing_error := arctan(COMBINED_POS_ERROR_M / (slant_range_m - COMBINED_POS_ERROR_M))

/// Requirement 4
output requirement_4
    /// Requirement 4a
    eval when slant_range > 14.0 with bearing_error > 0.8
    /// Requirement 4b
    eval when slant_range > 8.0 with bearing_error > 1.5
trigger ¬ requirement_4 "Bearing error threshold violation"
```

Fig. 6 Full specification