Towards Intelligent System Health Management using Runtime Monitoring

Christoph Torens[*] Florian-Michael Adolf[†]

DLR (German Aerospace Center), Institute of Flight Systems

Braunschweig, 38108, Germany

Peter Faymonville[‡] Sebastian Schirmer[§]

Saarland University, Reactive Systems Group

Saarbrücken, 66123, Germany

System health management is an important feature of autonomy, enhancing consistency checks, overall system robustness and even some degree of self-awareness. Seemingly unrelated, debugging and analysis of such complex systems is another challenge during development that should not be underrated. We propose that the so-called runtime monitoring of relevant properties and system requirements is a viable technique to support both aforementioned concepts. A suitable monitoring approach for a cyber-physical system has to be efficient and capable of supervising various specifications, possibly relating different data sources and data history. We present a formal approach for log-analysis and monitoring for the DLR ARTIS framework using the stream-based specification language LOLA, currently developed at Saarland University, for the runtime monitoring of formal specifications. We have evaluated this approach by specifying relevant properties as LOLA stream equations. While we have identified a number of possible improvements in the specification language, we have demonstrated, even with the current language, that online and offline monitoring of relevant properties is indeed possible and gives engineers a powerful tool for debugging as well as implementing health management concepts.

I. Introduction

The DLR operates ARTIS (Autonomous Research Testbed for Intelligent Systems), a fleet of unmanned aircraft for the research of functions towards all aspects of autonomous flight. As the name suggests, the research focus emphasizes on highly automated functions and autonomy. To this end, ARTIS utilizes several classes and sizes of aircraft with different autonomous capabilities as a generic flying platform for autonomy research.

One important aspect of autonomy is the capability to create high-level mission plans and, as part of this, generate paths from a given point to a desired destination. Since 2006, ARTIS has been equipped with a Mission Planning and Execution (MiPlEx) software framework that comprises real-time mission plan execution, 3-D world modeling as well as algorithms for combinatorial motion planning and task scheduling. In the remaining work, this system will be used to demonstrate applications of runtime monitoring.

The framework is based on a decoupled approach for path planning, trajectory generation, trajectory following, and inner loop flight control.[1] The rotorcraft's guidance algorithm has been evaluated in flight

[*]Research Scientist, German Aerospace Center (DLR), Institute of Flight Systems, Dept. Unmanned Aircraft, Lilienthalplatz 7, Braunschweig, 38108, Germany, AIAA Member.

[†]Research Scientist, German Aerospace Center (DLR), Institute of Flight Systems, Dept. Unmanned Aircraft, Lilienthalplatz 7, Braunschweig, 38108, Germany, AIAA Senior Member.

[‡]Graduate Student, Saarland University, Department of Computer Science, Reactive Systems Group, Campus E1 1, Saarbrücken, 66123, Germany.

[§]Student, Saarland University, Department of Computer Science, Reactive Systems Group, Campus E1 1, Saarbrücken, 66123, Germany.

tests with respect to closed-loop motion planning in obstacle rich environments. The work in Ref.[2] describes a control architecture behind the guidance layer. It achieves hybrid control by combining the main ideas from a behavior-based paradigm[3,4] and a three-tier architecture.[5] The behavior-based paradigm reduces system modeling complexity for composite maneuvers (e.g. land/take off) as a behavior module that interfaces with the flight controller, Fig. 1. The three-tier architecture has the advantage of different abstraction layers that can be interfaced directly such that each layer represents a level of system autonomy.

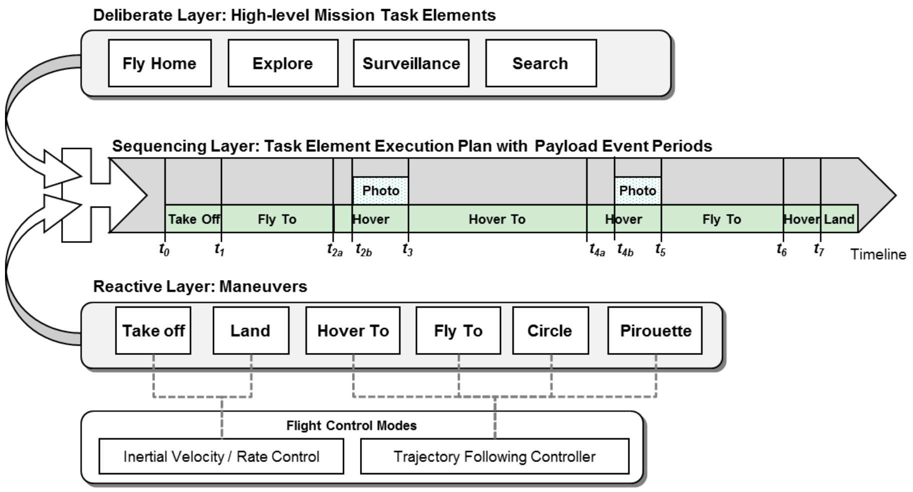

Figure 1. Mapping mission shown in context with the control architecture: High-level behaviors use task specific planners (Deliberate Layer), behaviors are compiled into plans (Sequencing Layer), and movement primitives (Reactive Layer) interface with the flight controller.

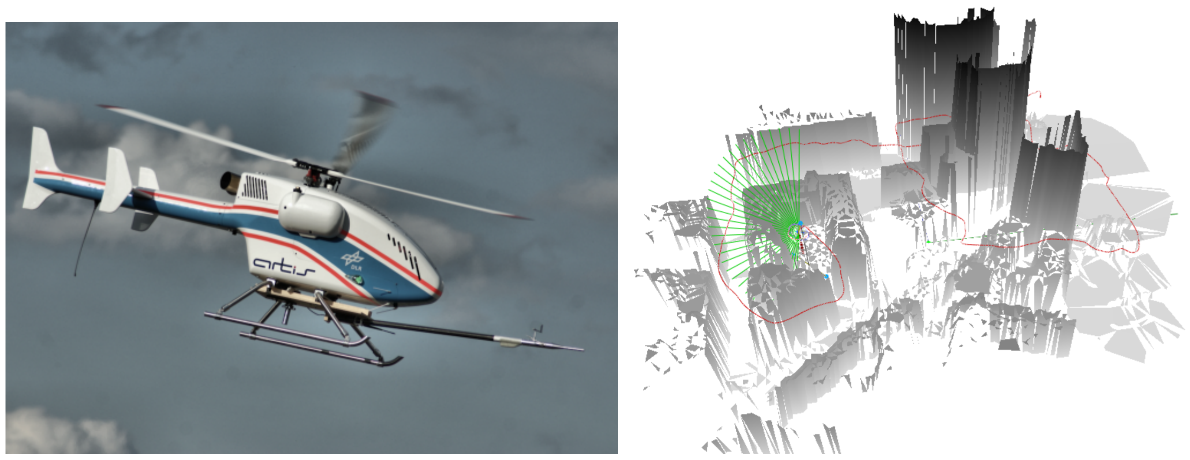

Figure 2. A DLR unmanned rotorcraft (left) and a simulation result of online planning in urban terrain for our rotorcraft with onboard perception (right): path flown (red), virtual distance sensor (green), obstacles mapped (grayscale).

As a result, the mission planner is able to execute autonomous behavior, for example exploration of unknown terrain as shown in Fig. 2.

The novel aspect of unmanned aircraft as an application is the missing onboard pilot, an aspect the public is most concerned about. Increased effort in verification and validation activities is therefore required. The overall goal is to achieve fail-safe behavior and to guarantee certain limits in the safe behavior of high-level software components as the mission planner.

A. Problem Statement

We want to discuss two main challenges for the development of an unmanned aerial vehicle (UAV), that first seem to be independent. At first with complex systems, there is always the need to have powerful methodologies to debug and analyze the system. For the ARTIS framework, extensive data from all sensors and software modules is logged into files. This logging capability is an important feature for debugging. However, going manually through system log-files to analyze system behavior can become a huge effort and is prone to errors. Analyzing more complex properties can become infeasible if data has to be derived, set in context to data history, or data from a different data source. Therefore, a tool automation for finding, filtering, or tagging of specific data with supplementary information can be a huge benefit for the analysis of logging data.

Secondly, future unmanned aircraft are expected to be highly autonomous. Besides functional capabilities, such as obstacle sensing and mission planning, another key issue of autonomy is the concept of health management. Health management enables the aircraft to assess the own capabilities and, in case of degradation, enables to react in a robust fashion by triggering contingency procedures. The first part of such a health management concept is the monitoring of the system status, to enable a form of self-awareness. The so-called Autonomy Levels For Unmanned Rotorcraft Systems (ALFURS) framework[6] is used to assess autonomy of unmanned aircraft, from Level 0, remote control, up to Level 10, fully autonomous. Starting from autonomy Level 3, required capabilities include health diagnosis and detection of hardware and software faults. In short, to achieve the concept of autonomy, an awareness of the system itself and its internal states is necessary to cope with abnormal system states, degraded situations and unforeseen environmental events.

B. Approach and Paper Outline

The concept of runtime monitoring is capable of both, offline analysis of log-files for debugging purposes, as well as the online supervision of system states and violation of specified properties to support health management. The use of a formal description language for the specification of properties enables formal reasoning and allows the automatic generation of monitors. Therefore, DLR and Saarland University are cooperating to enable the detection of specification violations by using formal methods.

In this paper, we present a formal approach for log-analysis and monitoring for the DLR ARTIS framework using a tool for the stream-based specification language LOLA, currently developed at Saarland University, for the runtime monitoring of formal specifications. Runtime monitoring is a lightweight formal method to ensure the correctness of a given system at runtime, i.e. through observation with a given formal specification. Its main advantages are that it can be incrementally introduced into the system, it is applicable even when exhaustive verification methods such as model checking fail due to large state spaces, and the specifications are descriptive and express stateful properties which are easy to maintain compared to hand-written monitoring code. When monitoring along the system execution, alerts of property violations can be used to influence the behavior of the system under scrutiny, e.g. to change to a different set of sensors or a more conservative flight controller. The stream-based formal specification language LOLA is, however, not restricted to simply output a binary answer for every input trace (either the property holds or not), but is much more expressive: It can be used to express and compute statistical properties of the current trace and thus can monitor the current system performance, e.g. to accumulate the deviations of the planned flight path to the actual one. This expressiveness is also very useful for the offline analysis of log-files to calculate system performance measurements, which may be computationally too expensive to be calculated on-the-fly.

As the first phase of integration, LOLA is used to analyze aforementioned log-files. Existing sets of data from real test flights, as well as the possibility to create custom logging data from software- or hardware-in-the-loop simulations facilitates this integration. In a second phase, selected specifications are supervised via online monitoring. To do this, the monitoring has to be integrated into the ARTIS framework, however, detailed architectural considerations are not in the scope of this paper. Further work at DLR is researching architectural considerations of integrating monitoring into a system architecture and triggering contingency procedures for safety-related hazardous events.

Utilizing a stepwise approach for the integration of runtime monitoring, different stages of benefits to system development and the system itself can be identified. A more detailed discussion of these benefits will be discussed in the next section:

1. Offline analysis of given log data to generally support logging and debugging, as well as facilitating complex analysis of internal processes.

2. Offline analysis of execution performance to assess specific metrics and correlate this to historical data to be able to supervise performance degradation.
3. Online monitoring of system boundaries and components states to support and increase the general situational awareness.
4. Online monitoring of safety requirements as an inherent part of the safety concept, e.g. to inform the pilot about abnormal states or to enable assurance cases of high-level runtime behavior.
5. Online monitoring of system components to invoke a degraded state or contingency procedure to increase robustness as part of health management.

The remainder of this article is structured as follows: The following section II presents aspects of runtime monitoring and system health management, in which aforementioned applications of runtime monitoring are discussed in more detail. Next, in Section III, related work is presented on runtime monitoring and system health management. Afterwards, Section IV presents the specific properties of the stream-based specification language LOLA. Then, Section V discusses the specific approach of runtime monitoring for our ARTIS unmanned aircraft. Finally, Section VI concludes this paper.

II. Runtime Monitoring and System Health Management

Runtime monitoring describes a collection of approaches to evaluate formal specifications on traces of systems in order to verify the correctness of the system. Two main modes of operation are distinguished: *online monitoring* and *offline monitoring*. In online monitoring, the interface of a system is observed at runtime one event at a time, and the monitor produces a verdict with respect to the specification according to the trace of observations seen so far. In contrast, for offline monitoring we may assume that the trace is immediately fully available and may be traversed in either direction, which allows for more efficient algorithms. Runtime monitoring is a lightweight formal method, compared to exhaustive verification methods such as model checking, which may be infeasible to apply for complex practical systems. Gradual application of runtime monitoring to existing systems is possible, since there is no need to first formally re-engineer the system to a model. Specification languages for monitoring can be more expressive (types, functions, . . .), since we can evaluate them directly on the given trace. Also it is possible, that verdicts of the monitor can be used in a feedback loop to influence the system behavior itself. Correctness of monitoring algorithm and specification is easier to argue than correctness of system under test. Since the same methodology is used, a single specification can be used for both online and offline monitoring. Additionally, the formal specifications can potentially later be used for model checking of components. And statistical analyses can be used to evaluate different runs of the system.

A. Application of Runtime Monitoring for Debugging and Behavior Analysis

Analyzing complex properties can become infeasible for debugging purposes. This is especially true, if data from log-files has to be derived to be able to analyze it, e.g. put into a calculation. It may also be the case, that such data needs to be set in context to data history, e.g. value A must not exceed a certain threshold for more than B seconds. In a similar fashion, data may need to get related to a different data source, e.g. if sensor S exceeds a threshold, then property T must always hold. An important feature is to filter out log data, according to requirements and tag it with a specific keyword for further analysis. Moreover, often an expert has to look at such system behavior and has to use his *gut feeling* to assess such data. In such cases it is not always directly possible to formalize the specific property, or the property is specified, but the specific boundary and threshold values are unknown. One possible solution to formalize such a *gut feeling* is the ability to find and compare specific values or patterns in previous, historic data. Therefore, it is necessary to easily check existing historical data for boundary values to come up with formalizations. Runtime monitoring can easily be used to filter out specific values or patterns and, for example, tag maximum or minimum values.

B. Application of Runtime Monitoring for Benchmarking and Identifying Performance Degradations

During the lifetime of a UAV parts may wear out. In some cases this can result in an immediate malfunction, in other cases this could be the cause for a performance degradation of the vehicle. Runtime monitoring can be used as a tool for benchmarking specific properties, to compare the performance to historical data or estimated behavior. Therefore, it is possible to automatically identify performance problems and possibly identify system failures before they become hazardous.

C. Application of Runtime Monitoring for Situational Awareness of a pilot

Health monitoring is an important aspect of modern aircraft. In manned aircraft, some of these monitoring functions are automated, others, maybe more complex monitoring functions are, however, left to the pilot. UAV clearly differ from manned aircraft with respect to the pilots distance to the aircraft, his situational awareness as well as his ability to bring resilience to unforeseen events into the guidance and control loop. The onboard pilot is replaced by a command and control data link that sends specified data from and to a safety pilot sitting at a ground control station. Therefore, the safety pilot may notice problematic situations or abnormal behavior of the aircraft only after it is too late to recover. Because of these reduced capabilities, software has to take over functions which the pilot would usually perform onboard, resulting in software systems of increasing complexity. Moreover, software has to take over supervisory functions which the pilot would usually perform, not only active tasks as mentioned above. As such it is important to increase the pilots situational awareness. Runtime monitoring can easily be used to focus the attention of a pilot to a specific critical item that has passed a certain warning threshold. For example, the display of flight information would be always visible to the safety pilot, but this information would be highlighted with a signal color if a limit is exceeded. For other information, that is not needed in all circumstances, it would be possible to display the information only if a limit is exceeded.

D. Application of Runtime Monitoring for Robustness and Intelligent Behavior

The same information that can be given to a pilot to increase situational awareness can also be given to the system itself. But beyond increasing the situational awareness of a pilot, a runtime monitor for supporting autonomy and intelligent behavior should be able to not only supervise specific properties, but additionally correlate such information to the known system state. Runtime monitoring is a suitable technique for such tasks, since this new data of system inputs must be continuously assessed and correlated to the current system states, as well as its history. Furthermore, it is possible to intelligently change the system state according to specific inputs and thus react differently for the same situation. For example, when sensory inputs cannot be trusted in the same way as under optimal conditions, safety boundaries for velocity and obstacle distance could be increased. Or when an unusual rate of fuel consumption is registered, the mission plan could be optimized for minimal fuel consumption, or the mission could be cut short, or a return to base could be initiated.

E. Application of Runtime Monitoring for Safety

In order to be able to detect system and software faults, critical modules need to be continuously monitored. In some cases, it may be easy to assess a system fault, e.g. if a system does not react anymore, or specific hardware systems which themselves have some degree of failure detection could directly give this input to a dedicated monitoring module. For complex systems on the other hand, especially for software intense systems controlling certain behavior of the aircraft, it may not be evident if there is a failure. In these cases more complex analysis is necessary to assess a failure. Furthermore, a failure may be dormant, in such a case, the failure can only be exposed if one or more additional properties are active at the same time. With runtime monitoring, it is possible to uncover dormant failures and analyze states as well as development and trend of failure events, if according properties are properly monitored.

Generally, if the monitor assesses a system fault, then a mitigation action must be triggered. The monitor could either initiate an action by itself, reset and reactivate the faulty system, or deactivate it and activate a backup system. The backup system could have reduced capabilities but would just be able to maintain or ensure a safe state. Therefore, the monitoring of systems and subsystems is the key approach to design fail-safe software systems.

As a result, the DLR ARTIS MiPlEx software component takes over active tasks that an onboard pilot should normally perform. To complement this, the proposed runtime monitor component takes over the supervisory tasks of the onboard pilot. As such, the runtime monitor not only supervises mere functionality, but acts as an intelligent component that assures high-level decisions and actions which do not cause a catastrophic situation and which are consistent with known environmental conditions.

III. Related Work

With the concept of monitoring, more specifically runtime monitoring, we refer to a formal methodology. By using formal specifications, runtime monitoring offers techniques to generate a monitor for supervision properties. The specialty of this approach is that it enables very efficient monitors, that can be verified because they are automatically derived from a formal specification. The concept of runtime monitoring is related to the concept of model checking. A good explanation of runtime monitoring and specifically a differentiation to other verification techniques, such as testing and model checking is given by Leucker.[7] Barringer[8] proposes to use runtime analysis of log-files to introduce formal methods into the development process using the metaphor of a Trojan horse. No system integration is needed to analyze generated log-files. Therefore, there is no additional effort needed to introduce this technique. As a later step, the same supervision properties developed for offline analysis, as well as further analysis, can be integrated online into the system after a suitable logging capability has been integrated into the system. Further details for the formal analysis of log-files is given in.[9] The specifications for such monitoring properties can be arbitrary and can differ largely from one case to another. However, Robinson[10] proposes to use software requirements for the monitoring of the software execution to provide assurances on software behavior. Runtime monitoring specifically for checking safety, security, and other critical properties is proposed as runtime certification by Rushby.[11] Runtime certification proposes the monitoring of runtime assumptions to justify assurance cases and thus make assumptions explicit.

Lately, runtime verification has been applied at NASA in context of Software Health Management[12].[13] Synchronous and asynchronous observers are used to assess the system status against a temporal specification. The synchronous observers evaluate the specification, only based on the past events, at each time stamp to true, false, or maybe. If necessary, the asynchronous observes will refine these values to true or false with the help of future events. In order to decide the health of the system the system status is fed into a Bayesian network. The network is used to perform diagnostic reasoning and system analysis. Temporal properties are given to the network as input which means that it is possible without much effort to compile the network into an arithmetic circuit. On the other hand, having no specification language to express these higher level specifications requires an expert in designing the Bayesian network, i.e. dependencies of nodes and the conditional probability. Additionally, Bayesian networks require an underlying directed acyclic graph which prevents using bidirectional dependencies, e.g. between two health nodes. A translation from a high-level specification language to a Bayesian network could help to specify a network in a concise and efficient way. Despite these restrictions software health management could identify previously unknown faults in the aircraft control system (file system, signal handling, and navigation).

From a certification point of view, the guidelines for development of civil aircraft and systems[14] identifies monitoring as an alternative protective strategy to complement a proposed system architecture. The preliminary aircraft safety assessment or preliminary system safety assessment (PASA/PSSA) would identify the need for such an protective strategy to validate that the architecture can be expected to meet the safety requirements.[15] Furthermore, the use of formal methods is standardized with the latest revision of the standard for software development for civil aircraft and its specific supplement for the use of formal methods[16]

As a result, the application of runtime monitoring for system health management gives an interesting perspective on monitoring properties, assuring behavior, and even argue for certification of unmanned aircraft.

IV. LOLA: Stream-based formal specifications

The formal specification language LOLA,[17,18] originally developed for monitoring synchronous circuits, is a typed language based on stream equations. A monitoring specification consisting of LOLA stream equations maps a given set of input streams to a set of intermediate and output streams and uses triggers to define user alerts based on the observed behavior. As stream types, LOLA supports boolean, string, integer, and double values and corresponding basic functions such as comparisons, arithmetic expressions and string matching. Especially the numerical types are useful since they facilitate the expression of quantitative statistics within

LOLA stream equations.

LOLA specifications are of a declarative nature. The output and intermediate streams are defined by an expression over input streams, constants, and output streams which allows to establish a correlation between several streams. Note that the equations only declaratively specify the value of an output stream, but do not fix an evaluation mechanism or an evaluation order, which is left to the monitoring algorithm. This allows us to leave optimizations, the management of monitoring state and the tracking of intermediate values to the monitoring algorithm. The same specification might be evaluated differently in the context of online or offline monitoring, where we have random access to the input trace. Furthermore, the ability to introduce intermediate output streams offers a natural way to structure and modularize the specifications. Thereby, the readability of LOLA specifications is improved and redundant stream computations are avoided. All these characteristics keeps the specifications succinct and expressive in comparison to monitoring via hand-written monitor code or temporal logic formulas.

Recall that a LOLA specification is given as a set of stream expressions. In order to be able to express complex temporal properties, the stream expressions may not only depend on the current values of other streams, but may also access past and future values of other streams, as can be seen in the examples below. If the specification of a stream contains dependencies on future stream values, the evaluation will be delayed until all dependencies can be resolved.

In comparison to classical monitoring of temporal logics like LTL,[19] LOLA allows quantitative statistics, modularization, and richer verdict domains in terms of the continuously produced output streams. The disadvantage of LOLA is the non-constant space complexity in the length of the trace for future dependencies.

Formal LOLA specifications are one way to implement monitoring into the system. Another possibility is to include hand-written monitors within the system. The difficulties of this approach, in comparison to an external monitoring approach like LOLA, are the violation of the principle of unobtrusiveness and the maintenance of the monitor code along the program.

The basic LOLA online monitoring algorithm is based upon two sets of equations, called stores: the resolved store R and the unresolved store U. All resolved equations will be in R and all unresolved equations will be in U. After new input values arrive, their values are added to R and propagated to all depending stream equations in U. If a depending stream is resolved, it is removed from U and added to R. Entries in R which are not required anymore are removed in time.

V. LOLA applications for ARTIS

In this section, we show some applications of runtime monitoring for UAV and provide examples for specifications, suitable for analysis of UAV. After a short motivation for the example, the according specification is listed and explained.

SUPERVISING SIMPLE BOUNDARIES In Listing 1, we show exemplarily how we can encode height boundaries into a LOLA specification. In our example, the height is restricted by an absolute bound, e.g. a legal restriction (Flight space, Line 3) and additionally by a chosen relative mission specific bound, e.g. never increase the height by more than 100 meters above takeoff altitude (Mission restriction, Line 7). Therefore, our LOLA specification requires only one `input` stream in Line 1 which represents, in each step of the stream, the current height of the UAV. Line 3 demonstrates how easy we can deploy our first absolute boundary. A `trigger` describes a signal to the user with a condition, i.e. `height > 150`. Notice that in this case, the output is only binary, i.e. either it fulfills the restriction or violates it. However, the full expressiveness of LOLA is apparent when specifying the relative height bound. In Line 5, the output stream `max_height` computes, at each step, the maximum relative height increase. We are taking the `max` out of the *previous* computed `max_height` and the current height increase, i.e. the difference between current height and starting height. Hence, the output stream `max_height` carries statistical information. Its definition incorporates the two most powerful LOLA operators, the absolute and relative offset operator. Both are used to express temporal dependencies between streams. The absolute offset operator, i.e. `height#[0, 0.0]` is used to capture the starting height and the relative offset operator accesses the last computed `max_height`, i.e. `max_height[-1, 0.0]`. The operators differ in the semantic of their first value in brackets. The absolute operator only allows a positive integer which represents the absolute position in the trace, e.g. here the first height value. The relative operator allows both negative and positive integers which are evaluated relative to the current position, i.e. a negative integer represents a past value of the stream and a positive integer a future value. The second value in brackets specifies the out-of-bounds value, i.e. the value which is used whenever the accessed position is not existing or never exists, e.g. accessing a past value at the first position of the stream. In Line 7, we specify a trigger on the `max_height` to notify the user that the mission restriction is violated. In case the trigger condition is satisfied, the user receives the message *Mission Restriction Violated*.

```
1   input double height
2
3   trigger height > 150.0 with "Absolute Altitude Flight Space Restriction Violated"
4
5   output double max_height := max ( max_height[-1, 0.0], height - height#[0, 0.0] )
6
7   trigger max_height > 100.0  with "Relative Altitude Mission Restriction Violated"
```

Listing 1. A simple LOLA specification which checks that no absolute and relative height restrictions are violated during the flight.

Incorporating redundant sensory input A UAV uses several redundant sensors to guarantee the detection of contradictions in case of erroneous behavior. In Listing 2, we indicate how LOLA can be used to guarantee safe maneuver of the UAV. Our `input`s are the range of sight of a laser and the range of sight of an optical camera sensor as well as the commanded velocity. For a real pilot, it is natural to only fly depending on the current range of sight. The following specification checks whether the UAV mimics this natural behavior. In Line 3 and 4, we introduce two constants which are used as (arbitrary) bounds to either raise a warning or an error. A warning can be used by a controller to adapt the commanded velocity to the current sight whereas an error aborts the mission. Note that constants allow encapsulation, i.e. changing the constant once instead of replacing each occurrence of the value. The output stream on which the warning and error is raised is given in Line 6. It calculates the difference between the minimal range of sight of the sensors and the absolute commanded velocity, i.e. it raises a trigger whenever we command a velocity which is too fast given the current range of sight. This is an example for the intelligent adaptation of high-level system behaviors according to inputs from other subsystems, here the different vision sensors. Monitoring is capable to identify situations where the capabilities of the vision system are reduced, compared to optimal conditions, and thus facilitates to adapt the safety thresholds accordingly.

```
1   input double visualRange_laser, visualRange_optical, velocity
2
3   const double velocity_warning := 5.0
4   const double velocity_avoid   := 2.0
5
6   output double  difference :=  min ( visualRange_laser, visualRange_optical )  -  abs(velocity)
7
8   trigger difference < velocity_warning  with "WARNING: Velocity Limit reached"
9
10  trigger difference < velocity_avoid    with "ERROR: Abort mission."
```

Listing 2. A LOLA specification which receives three inputs: the range of sight of a laser camera sensor, the range of sight of an optical camera sensor, and the current velocity. Based on the current minimal range of sight and the velocity a trigger is raised which depicts either a warning or an error.

Checking high-level behavior In Listing 3, a specification is displayed which considers a possible walk of the mission manager on its transition system and checks its correctness. Here, the specification depicts only the basic idea and does not cover the complete transition system of the underlying model. In practice, the coverage of the complete transition system is possible and abstractions from some implementation details are applicable. In Listing 3, the specification states that whenever the `safety_pilot` is set, after three consecutive time steps (measured at 25Hz), the mission manager (i.e. the autopilot) has to be turned off after 120ms. When the `safety_pilot` is not set or if it gets deactivated, the manager should remain on. The `safety_pilot[1..3, false, &]` operator is called windowing. The operator can be unfolded into several relative offset operators which are linked by the connector given by the third argument in brackets, e.g. '&'. The above statement can be unfolded to: `safety_pilot[1, false]` & `safety_pilot[2, false]` & `safety_pilot[3, false]`. In our example, the windowing is based on future values and, in case of a positive condition evaluation, we restrict the future behavior of our `ManagerState` to *ManagerStateOff*. If this restriction is violated, `turned_off` would evaluate to true and raises a trigger notification, defined in Line 9. Notice that, similar to Listing 1 where we computed the maximum height increase during the flight, we could also compute a statistic on the responsiveness of the `ManagerState`. For instance, we could reformulate the example to capture the occurred worst case time required to switch the manager off when the `safety_pilot` is set for three consecutive time steps. Afterwards, a trigger on this worst case time could be specified to notify a violation to the user. This re-use of statistical streams in other computations or notifications are the key feature of the modular composition of LOLA specifications.

```
1  input bool safety_pilot
2  input string ManagerState
3
4  output bool  turned_off := if safety_pilot[1..3, false, &]
5                                 { ManagerState[3, ""] = "ManagerStateOff"  }
6                             else
7                                 { ManagerState = "ManagerStateOn" }
8
9  trigger turned_off
```

Listing 3. A LOLA specification which states that whenever the `safety_pilot` is set for three consecutive time steps then the `ManagerState` should be set to *ManagerStateOff*.

Filtering and tagging of data So far, the user receives notifications in case a trigger condition is satisfied. The technique we present here allows to generate a log-file as a LOLA evaluation outcome. LOLA can be used for automatic filtering and tagging of data according to specific properties. These techniques especially support the post-flight analysis. Filtering is useful to separate meaningful data from the streams and tagging is used to add further information (tags) to the data. The amount of data that needs to be analyzed manually can be reduced and the attached tags allow to add additional explanatory information that can further support manual analysis.

In Listing 4, an example specification is depicted. The input streams represent the current velocity and the current state of the mission manager. Two output streams are specified. The stream `cnd` is used as condition when to log the data into the new generated file at location *filteredFlight.log* (Line 8+9). The value of the output `magnitude` can be seen as an indicator how much the velocity exceeds the specified threshold. The function `floor` rounds down its argument. The semantic of `tag` is: If `cnd` holds then the stream values of the specified streams after *with* are written in the respective column, specified after *as*. Notice that the columns of the new file can be named arbitrary, e.g. the new first column is called *velocity*. Listing 5 illustrates this approach. On the left side the incoming data streams are depicted and on the right side the generated log-file is shown. The first line indicates the respective stream and the following lines its data values. For instance, the stream *vel* contains the following values given in the correct temporal order: 10, 15, 8, 24, and 30.

```
1  input double vel
2  input string ManagerState
3  const double threshold := 10.0
4
5  output bool cnd      :=  vel > threshold
6  output int magnitude :=  floor( vel / threshold )
7
8  tag as velocity, state       ,   magnitude   if cnd
9    with vel     , ManagerState,   magnitude   at "filteredFlight.log"
```

Listing 4. A LOLA specification which generates as evaluation outcome a new log-file. The data logged in the new file is a filtered version of the incoming data and it is additionally tagged by a value indicating the magnitude the threshold is exceeded.

```
vel      ManagerState
10       "Accelerate"
15       "Slow-down"
8        "Accelerate"
24       "Accelerate"
30       "Remain"
```

```
velocity     state           magnitude
15           "Slow-down"     1
24           "Accelerate"    2
30           "Remain"        3
```

Listing 5. On the left side two streams are depicted: the current velocity and the state of the mission manager. The first line represents the streams and the following lines the values of the respective stream in the column, temporally ordered. On the right side, the resulting log-file is shown when applying the specification in Listing 4 to the streams given on the left side.

VI. Conclusion and Outlook

The ARTIS UAV research platform enables the development and test of autonomous software functions. In particular, two main challenges for the development of a UAV were considered. As one challenge, the need of powerful methodologies to debug and analyze the system behavior via huge amounts of logged data. Automated techniques are necessary to cope with the growing system complexity, since manual data inspection takes huge effort and is prone to errors. The second challenge is the implementation of capabilities to assess the own system status, to detect system failures, and enable intelligent system health management. This self-awareness plays an important role in autonomous systems since it is necessary for robust reactions in degraded situations.

Runtime monitoring can be used to cope both aforementioned problems. We have used the ARTIS UAV research platform to demonstrate our best practice approach. As monitoring specification language, we chose LOLA. LOLA is based upon streams and closes the gap between temporal logics like LTL and hand-written monitors. Streams can incorporate mathematical computations such that quantitative statistics are supported and also allows to express temporal (past *and* future) connections between other streams and their values. Further, the modular composition of LOLA helps to keep the specification concise and human readable. Given this innovative expressiveness of the used formal language, the utilized tool for the specification language LOLA enables us to specify a large variety of specifications and we were able to research interesting properties for the runtime monitoring of UAVs. The first application was focused on offline log-file analysis which offers a first step to integrate monitoring into the existing development processes. This approach supports the debugging and analysis of log files, that was mentioned as the first challenge in the problem statement. The other applications depicted an incremental approach towards an intelligent component that assures high-level decisions and actions to support the second problem for engineering and developing an autonomous UAV. In general, a full online integration of runtime monitoring into the system gives powerful possibilities to advance system awareness, health management, and intelligent behavior. An online runtime monitor can supervise high-level behavior, safety, performance, and situational awareness. Therefore, runtime monitoring can be utilized as a tool to increase autonomy and make software-intense systems more robust and fail-safe.

Future work will focus on refining the approach and implementation of our online monitoring framework, specifically applications for robustness and safety, as well as extending the specifications being supervised. Finally, further research about the triggering and scope of contingency procedures and architectural considerations to include monitoring and contingency management functions into a system architecture are necessary, so that these functions are independent, unobtrusive, responsive, and realizable.

References

[1]Lorenz, S. and Adolf, F. M., "A Decoupled Approach for Trajectory Generation for an Unmanned Rotorcraft," *Advances in Aerospace Guidance, Navigation and Control*, edited by F. Holzapfel and S. Theil, Springer Berlin Heidelberg, 2011, pp. 3–14, 10.1007/978-3-642-19817-5_1.

[2]Adolf, F. and Thielecke, F., "A Sequence Control System for Onboard Mission Management of an Unmanned Helicopter," *AIAA Infotech@Aerospace Conference*, 2007.

[3]Brooks, R. S., "A robust layered control system for a mobile robot," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990, pp. 204–213.

[4]Flanagan, C., Toal, D., Jones, C., and Strunz, B., "Subsumption Architecture for the Control of Robots," *Proceedings Polymodel-16*, Sunderland, UK, 1995, pp. 150–158.

[5]Bonasso, R. P., Firby, J., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M. G., "Experiences with an Architecture for Intelligent, Reactive Agents," Vol. 9, April 1996, pp. 237–256.

[6]Kendoul, F., “Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems,” *Journal of Field Robotics*, Vol. 29, No. 2, 2012, pp. 315–378.

[7]Leucker, M. and Schallhart, C., “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, Vol. 78, No. 5, 2009, pp. 293–303.

[8]Barringer, H., Groce, A., Havelund, K., Rydeheard, D., and Smith, M., “Runtime Verification of Log Files, a Trojan Horse for Formal Methods,” 2009.

[9]Barringer, H., Groce, A., Havelund, K., and Smith, M., “Formal Analysis of Log Files,” *Journal of Aerospace Computing, Information, and Communication*, Vol. 7, No. 11, 2010, pp. 365–390.

[10]Robinson, W. N., “Monitoring software requirements using instrumented code,” *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, IEEE, 2002, pp. 3967–3976.

[11]Rushby, J., “Runtime certification,” *Runtime Verification*, Springer, 2008, pp. 21–35.

[12]Reinbacher, T., Rozier, K. Y., and Schumann, J., “Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems,” *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 8413 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, April 2014, pp. 357–372.

[13]Schumann, J., Rozier, K. Y., Reinbacher, T., Mengshoel, O. J., Mbaya, T., and Ippolito, C., “Towards Real-time, Onboard, Hardware-supported Sensor and Software Health Management for Unmanned Aerial Systems,” *International Journal of Prognostics and Health Management*, 2014.

[14]The Engineering Society For Advancing Mobility Land Sea Air and Space, “4754A - Guidelines for Development of Civil Aircraft and Systems,” 2010.

[15]The Engineering Society For Advancing Mobility Land Sea Air and Space, “ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment,” 1996.

[16]RTCA, “DO-333/ED-216 Formal Methods Supplement to DO-178C and DO-278A,” 2011.

[17]D’Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., and Manna, Z., “Lola: Runtime Monitoring of Synchronous Systems,” *12th International Symposium on Temporal Representation and Reasoning (TIME 05)*, IEEE Computer Society Press, June 2005, pp. 166–174.

[18]Faymonville, P., Finkbeiner, B., Schirmer, S., and Torfah, H., “A Stream-Based Specification Language for Network Monitoring,” *Runtime Verification*, Springer Nature, 2016, pp. 152–168.

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