UNIVERSITÄT
DES
SAARLANDES

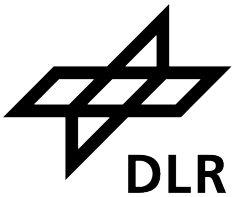
Mathematics and Computer Science
Department of Computer Science

Master's Thesis

Runtime Monitoring with LOLA

Schirmer Sebastian
Saarbrücken,
November 11, 2016

In cooperation with:



Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

Institute of Flight Systems
Department of Unmanned Aircraft

Supervisor:
Prof. Bernd Finkbeiner, Ph.D.
Reactive Systems Group
Saarland University
Saarbrücken, Germany

Advisors:
Peter Faymonville, M.Sc.
Reactive Systems Group
Saarland University
Saarbrücken, Germany

Dipl.-Inform. Christoph Torens.
Institute of Flight Systems
German Aerospace Center
Braunschweig, Germany

Reviewers:
Prof. Bernd Finkbeiner, Ph.D.
Reactive Systems Group
Saarland University
Saarbrücken, Germany

Prof. Dr.-Ing. Stefan Levedag
Institute of Flight Systems
German Aerospace Center
Braunschweig, Germany

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

Date Signature

Abstract

Runtime monitoring is a formal method for analyzing system executions. This analysis improves the confidence in the behavior of the system, either by improving the comprehension of the system or by checking the adherence of desirable properties. Monitoring can be used offline based on log files but also online along with the system being executed. The latter facilitates feedback at runtime. A stream-based specification language for the desirable properties is Lola. Given a set of input streams, a set of output streams is evaluated. Lola is kept simple and expressive and, hence, closes the gap between temporal logic and hand-written monitor code.

The DLR ARTIS framework is used for research on autonomy concepts, applications, and implementations for unmanned aircrafts. Important aspects of increasing autonomy involve correctness, safety, robustness, and system health management. In all of these aspects, runtime monitoring is a useful method to support the task of their implementations.

In this thesis, the applicability of Lola in the context of unmanned aircraft is elaborated. Based on interviews with DLR engineers, desirable properties are formalized in Lola specifications. In addition, the main contribution of this thesis is to adapt Lola to the domain requirements. Therefore, the Lola specification language is extended by new operators to increase its usability and expressiveness. For offline monitoring, existing logged flight data is analyzed and the usage of Lola in practice is examined. For online monitoring, DLR's available software-in-the-loop and hardware-in-the-loop simulations are used to evaluate the impact of monitoring on the system. In both monitoring cases, Lola is efficient and fast enough and, thus, can be used in practice. The specifications show that Lola is capable of expressing required properties. Further, in the online experiments, the effect of Lola on the system is hardly measurable.

Acknowledgments

I am very grateful to Prof. Bernd Finkbeiner for both my student assistant job, in which I got the opportunity to acquire programming experience by implementing Lola, and for the exciting challenge to actually deploy it in practice. I also want to thank Peter Faymonville for his valuable support, constructive criticism, and guidance during the development of the implementation and the thesis.

Moreover, I would also like to thank all DLR employees for their time, explaining topics of their research areas to me. I am grateful to Florian-Michael Adolf for his enthusiasm towards monitoring, Susanne Schulz for creating the detailed plots, and Danilo Galisteu for the support I received during the HiL simulations. Especially, I am thankful to Christoph Torens who I could always address in case of further questions and advice.

Furthermore, I would like to thank Prof. Bernd Finkbeiner and Prof. Stefan Levedag for reviewing this thesis.

Finally, I am very grateful for the support of my family and close friends. A special thank-you goes to Jesko Hecking-Harbusch and Marcel Maltry for not only proof-reading this thesis but also for letting me face the end of my study period with a smile in one eye and a tear in the other.

Contents

1 Introduction

The name of the specification language LOLA is motivated by the movie "Run Lola Run". "Run Lola Run" is a 1998 german action movie written by Tom Tykwer in which the main character is called Lola. In the beginning of the movie, Lola receives a phone call from her boyfriend in which she gets told that she needs to obtain 100.000 Deutsche Mark to save his life. After the call, the story is divided into three story lines each starting at this point in time. In each story line, Lola tries different strategies to obtain the money. Each strategy is based on gathered information from previous tries. Using this past information, she finally manages to rescue her boyfriend.

In reactive systems where the system has to possibly make safety-critical decisions based on the environment, information about the past can support such decisions. *Runtime monitoring* processes given information to identify violations of important properties of the system. Using the monitor outcome the system can alter its behavior to mitigate critical situations.

In runtime monitoring, a *specification* of desired system properties is given and a *monitor* checks whether the system execution fulfills it. The events of the execution are either given to the monitor using system instrumentation at runtime or by reading a logged data file. The former is called *online monitoring* and the latter is called *offline monitoring*. In both, as outcome, the monitor outputs *verdicts* which indicate the adherence of the execution to the specification. The verdicts for the execution may, in addition to property satisfaction (*true*, *false*), also be of analytical nature (e.g. average). Both facilitate powerful feedback on the system where especially the analytical feedback helps to refine and to improve the understanding of the decision-making of the system. The monitor *may* provide feedback to the system at runtime or the verdict can be interpreted elsewhere, e.g. by a human. Figure 1 illustrates the approach.

Runtime monitoring is a formal method allowing to provide mathematical guarantees on the reliability of systems. We distinguish two methods: *dynamic* and *static* verification. Static verification refers to the analysis of the system prior to its execution. An example is model checking. Similar to runtime monitoring, in model checking a specification is given but also a model of the system under scrutiny. Then, *all* executions of the model are checked against the properties of the specification. The resulting correctness guarantee on all executions is highly desirable but unfortunately, in practice, model checking is often not applicable. The reason for that is that model checking suffers from the state-space explosion problem which turns it beyond current computational capabilities for more complex practical systems. Runtime monitoring falls under *dynamic* verification where only a single execution of the system is under scrutiny. As such, we are not able to argue about the complete correctness of the system but instead about the cor-

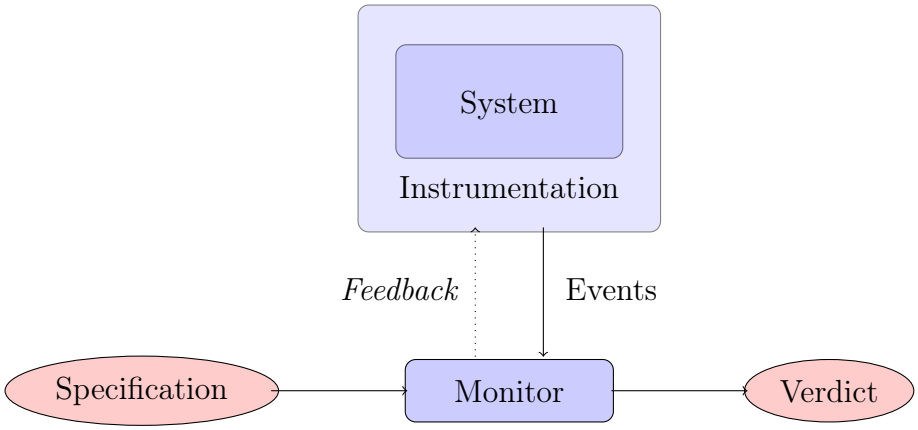


Figure 1: Overview of the online monitoring process.

rectness of the current execution run. However, by considering a single execution we are able to express more complex (analytical) properties since we can directly evaluate them during execution.

As a result, the following questions arise: What are the boundaries of the expressiveness? What properties are expressible? Are computationally expensive properties really required? Can we monitor them efficiently along the execution trace in such a manner that the system is unaffected?

In general, monitoring should be a method to verify properties *of the system* and not a method for solving computational problems *within the system*. Due to the fact that the specification language adapts to required properties, it is often called *domain-specific*.

In this thesis, we focus on utilizing the specification language Lola in the context of unmanned aerial vehicles (UAV). Aerial vehicles are highly complex systems which react autonomously. This autonomy implies that the UAV is capable of perceiving the environment which involves handling and reasoning about large data sets. Additionally, UAVs are safety-critical and, therefore, they have to adhere to strict regulations. The system behavior should be correct, secure, and reliable, even under severe conditions. In this thesis, we elaborate on whether runtime monitoring can be used to support this goal. More specifically, we decide whether runtime monitoring with Lola suits the requirements of this domain. We specify desirable properties based on interviews with engineers of the German Aerospace Center (DLR). Further, we consider the expressiveness of Lola but also the practical applicability, i.e. efficiency and usage. Considering the efficiency, we apply online and offline monitoring to evaluate the computational overhead on the system. Software and hardware simulations, available at the DLR, are used for online monitoring.

Outline

In Section 2, we give additional background information on the research area of runtime monitoring. In Section 3, we present the German Aerospace center, especially the research framework ARTIS, which supports the identification of interesting domain properties and also provides the UAV simulations used for the experiments. We present syntax and semantics of the Lola specification language in Section 4. In Section 5, we introduce domain-specific extensions to Lola. Section 6 provides attained specifications, gives some implementations details, and presents and discusses the experimental results. We conclude this thesis in Section 7 and address future work.

2 Related Work

In this section, we give a brief survey of runtime monitoring. We start with general notions followed by a presentation of related work, and other existing runtime monitoring frameworks.

Runtime monitoring is a formal method which allows for checking whether *a single execution trace* of a system under scrutiny satisfies given correctness properties [24]. These properties are formalized in a so called *specification* based on which a *monitor* is generated. Formally, in its simplest form the monitor has to solve the so-called *word problem*, a classical problem in automata theory, i.e. whether a given *finite* word (the trace) is included in the language (defined by the specification). Monitors are able to argue at runtime, or after the system has terminated. In offline monitoring, the complete trace of the system is already generated. Algorithms can increase the efficiency by utilizing the entire trace, e.g. by traversing the trace backwards. Typical areas of applications are trace analysis, debugging, and testing. In contrast, in online monitoring the execution is still ongoing and, thus, the trace is continuously extended. Consequently, the mentioned efficient algorithms may no longer be applicable. However, the great advantage of online monitoring is that the verdict of the monitor can give real-time feedback to a user or supervisor, or even the system itself. This can be achieved by feeding the verdict of the monitor back to the system which allows to mitigate a detected unintended system state.

Extensions of the Verdict Domain
Extensions of the verdict domain (*true* and *false*) were elaborated as follows. In [16], a 3-valued semantics for linear temporal logic (LTL)[35] called LTL_3 is defined. LTL is a popular specification language for trace properties. It is originally defined for infinite traces and hence some adjustments had to be made to argue over finite traces. In LTL_3 the two value semantics of LTL were extended by an additional verdict representing *inconclusive*, i.e. the current finite trace cannot be evaluated to *true* or *false*. In [17], the authors argued why LTL_4 is required for runtime monitoring. LTL_4 further distinguished the verdict *inconclusive* between *probably true* and *probably false*. An example why this separation is required for finite traces is the property *whenever there is a request, this request has to be granted eventually*. Previously in LTL_3, the verdict would always be *inconclusive* since neither exists a good nor a bad prefix for this property. LTL_4 allows to distinguish between cases where there is an open request (probably false) and cases without an open request (probably true). In [22, 26], the notion of *incrementally computable statistical measures* is introduced which provides an infinite verdict domain. There, the specification language (Lola) is extended by different types and functions and, hence is capable of computing statistics like the *average over an integer trace*.

Monitoring is based on a given specification. Advantages of a formal specification language are:

- The monitor is more concise and readable than hand-written code.
- Changing the specification and then automatically generating a new monitor is easier and faster than modifying a written monitor.
- Correctness of the specification suffices.
- Optimization in the monitor framework carry over automatically.
- Formal specifications allow to guarantee bounds on memory consumption.

In [24], a categorization of the different implementation approaches is given. A distinction is made between *external* and *internal* languages. *Internal* specification languages are embedded into an existing programming language. Often programming languages like Scala [9] are used as they already offer straight forward implementation of the required operators. On the other hand, *external* languages are stand-alone. The advantage of a stand-alone language is that optimizations can be implemented by choice.

Monitor System Integration
Another interesting aspect of runtime monitoring is the integration of the monitor into the system. How does the monitor receives probes of the internal system state in a sufficient way? This integration can be either *inline* or *outline* [24]. Inlining of a monitor instance means that the monitor is directly included in the system code [31]. Whereas outlining means that the monitor is an external entity [28]. In [47], the authors discuss different monitoring approaches. They conclude that hardware monitors have the advantage of non-intrusion but turn inapplicable for more complex systems. The main drawbacks of inlined or outlined monitors in software are the potential side effects and the resulting alternation of the target system. On the other side, instrumentation offers a wide access to system information. In hybrid monitors, the target process is instrumented in such a way that it emits events when interesting properties, e.g. variable values, change. These events will then be passed to an dedicated hardware monitor for its evaluation. Finally, they shortly discuss on-chip monitors which will reoccur in the next paragraph.

System Health Management
As of recent work, runtime verification has been applied at the National Aeronautics and Space Administration (NASA)[7] in context of System Health Management (SHM). In [28, 36, 39, 40], the different aspects of this approach are

elaborated. There, the authors present three system requirements for SHM which are crucial to monitoring the system health. First, *responsiveness* states that system faults have to be detected within reasonable time to allow mitigation. In order to achieve this, the system has to be monitored continuously. *Unobtrusiveness* requires the system properties, i.e. functionality, timing, certifiability, and tolerances, to remain unchanged despite the SHM framework integration. Last, the SHM must be *realizable*, i.e. plug-and-play manner for the connection of other systems and specifications.

In order to achieve these requirements they combine formal methods and a probabilistic graphical model in their framework rt-R2U2. The rt-R2U2 is implemented in hardware which runs externally and, hence, guarantees unobtrusiveness. As formal method, they use runtime monitoring of metric temporal logic (MTL) [14]. MTL is based on LTL and allows to specify time bounds on the temporal operators. To assure responsiveness, the MTL evaluation is based on synchronous and asynchronous observer pairs. The synchronous observers evaluate the specification, only based on past events, at each time stamp to true, false, or inconclusive. If necessary, the asynchronous observers will refine these values to true or false with the help of future events. They use field programmable gate array (FPGA) to meet the realizability requirement. FPGAs allow to reprogram the circuit in order to adapt to changes in the specification.

Afterwards, the outcome of the observer pairs is passed to a bayesian network (BN) [23] for the high-level health reasoning. The network is used to perform diagnostic reasoning and system health analysis. This approach does not require any specification for the health reasoning part since reasoning is performed with the help of BNs. Temporal properties are only implicitly given to the network via the observers which avoids the explicit and more complex reasoning over time in the BN. However, there is a break in the formalism. For a user, it would be beneficial to have a common formalism to specify both the temporal properties and the high-level reasoning. Not having such high-level specification language for the reasoning requires an expert for designing the BN, i.e. dependencies of nodes and the conditional probability. Additionally, BNs 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 BN could help to specify a network in a concise and efficient way. The rt-R2U2 framework was able to identify previously unknown faults in the aircraft control system.

Alternative Approaches

In [13], a specification language based on quantitative regular expressions (QREs) is presented. Regular expressions are a common way for describing regular languages in computer science. Regular languages are prominent because they cover

exactly the class of languages accepted by deterministic finite automata. QRE are quantified because they allow to specify numerical queries over a sequence of alphabet characters, called stream. Here, an alphabet character is a predicate over a domain, e.g. $\mathbb{D} = \mathbb{R} \cup \{end_h, end_d\}$, end_h and end_d indicate that an hour and a day ended, respectively. Possible numerical operators are for instance *min*, *max*, and *sum*. The framework follows a filtering and composition paradigm. Assume a property like *the average download rate per hour over the course of a day* for the given domain $\mathbb{D}$. When formalizing this property, we first need to specify a regular expression to identify whenever an hour ended: $r_d = \mathbb{R}^* \; end_h$. Next, we iterate over this filtered data to compute the average download rate in this segment: $f_1 :=$ iter-avg(r_d). Finally, split-sum$(f_1, end_d \; ? \; 0)$ takes the sum over the average download rates when the day ended.

Another monitoring approach is RULER [15]. RULER is a rule-based trace analysis tool. The specification is given as a set of named rules. These rules consists of a condition part and a consequence part. The condition part is a conjunction of predicates and the consequence part is a disjunction of conjunctive predicates. The predicates are restricted to be non-temporal, i.e. temporal dependencies will be encoded in rules. The predicates in the consequence part are called activated whenever their respective rule condition holds. Active predicates are non-persistent. They are only activated for the next evaluation step and then are automatically deactivated. The evaluation is based on a frontier set which contains all currently activated predicates. Given an event sequence over `CurInt` with an unique starting event `Start` its sum can be evaluated by the following rules:

`Init:`	`Start`	$\rightarrow$	{`CurSum(0)`}
`CurSum(i:int):`	`CurInt(k:int)` $\wedge$ `CurSum(i)`	$\rightarrow$	{`CurSum(i+k)`}

Rules are a common reasoning system in artificial intelligence. Therefore, it is interesting to identify the connection between the two research domains.

BeepBeep 3 [29] is a tool that was invented to close the gap between runtime monitoring and complex event processing (CEP). In contrast to runtime monitoring where a *property* over a trace is evaluated, CEP answers a *query*. It can be seen as a database problem. The author argues that CEP allows to manipulate data, i.e. computations, whereas most monitors provide only a boolean result. Further, CEP tools are more focused on these manipulations and lack in the expressiveness of temporal dependancies which is a core feature of runtime monitoring. BeepBeep 3 aims at combining computational and temporal expressiveness. BeepBeep 3 is a stream-based framework, similar to Lola. So called processors receive inputs and produce an output. The processor output can be piped to another processor, i.e. it acts as an input. Whenever all input traces obtain a value the processor produces its output. As a result of this processor architecture, it might happen that a processor has to build up a buffer for one input while waiting for another. The

tool allows for different ways of manipulating these input trace buffers, e.g. the decimator operators (returns only the n-th input event and removes the others) which is complemented by the freeze operator.

Note that, all presented approaches are event-triggered, i.e. they are invoked whenever an new event occurrs. An alternative approach is time-triggered behavior where the monitor is invoked periodically and reads the events. In [18], the authors argue that event-triggered monitoring can suffer under bursts of new events at runtime. As an example, they show how a C program can be augmented based on its control-flow graph such that a monitor which awakes periodically does not miss any changes. Furthermore, it is shown that the LTL_3 properties without the next operator can be soundly verified. Note that when augmenting the C program by further instructions, the runtime overhead may change. In addition, the responsiveness of the monitor depends on the wakeup period. The approach is implemented in the tool RiTHM [32]. In [49], an hybrid approach is proposed where depending on the current execution, the monitor can switch between a time-triggered and an event-triggered behavior.

3 Field of Application

In runtime monitoring, the specification formalism is often considered as domain-specific. The expected expressiveness of the formalism highly depends on the properties of the system under scrutiny. A restriction to the expressiveness of a formalism is the monitorability. The more expressive a formalism is the more expensive is monitoring properties of the respective formalism. Additionally, the property description should be easy to write and to read for a user. Criteria for this are for instance conciseness and elegance. In order to find a good tradeoff between expressiveness, monitor efficiency, and usability, it is necessary to identify families of required and interesting properties for the given domain.

In this section, we give a summary of our domain: the German Aerospace Center (DLR). First, we present the project for which we are going to integrate our monitor approach. Second, the participating ARTIS working groups are introduced. Then, we give an indication of the kind of data a monitor receives and should be able to handle. Afterwards, we shortly present the current state of the software development, verification, and validation. Finally, the problem of uncertainty of future restrictions and the resulting decision-making is outlined briefly. Additionally, in the last subsection we list the different possible applications of runtime monitoring.

3.1 German Aerospace Center

The German Aerospace Center[1][3], abbreviated DLR, is Germany's national research center for aeronautics, space, energy, transport, and security. The DLR was established in 1969 and nowadays consists of 33 institutes and facilities in sixteen national locations and four international offices with its headquarter in Cologne.

This thesis is the outcome of the collaborative work between the reactive systems group (Saarland University) and the department for unmanned aircraft (DLR - Braunschweig). The main research topic of the department for Unmanned Aircraft is the research of autonomy concepts, applications, and implementations for unmanned aircraft. Among others, flight control, sensor fusion, and mission control are active research areas to accomplish this task. The main research tool is the Autonomous Research Testbed for Intelligent Systems (ARTIS) framework with its ARTIS fleet of aircraft.

3.1.1 ARTIS

Starting with a single test aircraft the ARTIS fleet now consists of an entire fleet of aircraft of different types and properties. The testbed offers different degrees of

[1] Deutches Zentrum für Luft und Raumfahrt e.V.

autonomy such that an incremental approach towards complete autonomy can be applied. This approach is required since a higher degree of autonomy adheres an increasing software complexity which has to meet several constraints on software maintenance and guarantees towards code quality. Additionally, since aircraft are a safety-critical domain, strict regulations with high demands on correctness apply such that formal techniques are of interest. Unfortunately, aircraft are both safety-critical and highly-complex systems. One indication of the high complexity of the system is a wide range of sensors from simple GPS and magnetometer sensors to cameras and laser sensors for high-resolution image processing. Therefore, formal methods like model checking tend to turn out inapplicable for the complete system.

Hence, there is great interest in finding a sweet spot for the applicability of formal methods, for instance by using a compositional approach, i.e. splitting the overall system into several parts and applying dedicated methods to some parts and model checking with a higher degree of abstraction to others [19, 21]. However, this approach requires that both sides complement each other which is often complicated. Therefore, in this thesis, we chose to apply runtime verification as lightweight technique [24]. Runtime verification cannot guarantee the overall safety of the system but instead the safety of the currently observed trace, which has a lower complexity and can be seen as a first step towards system verification. Online feedback of the monitor can be used to improve the self-awareness and health of the system which improves the trustworthiness and safety of the system [41].

3.1.2 Involved Working Groups

The department of unmanned aircraft is divided into three working groups to cover different research areas. The three working groups are: Sensor Fusion and Environment Perception (SF), Flight Control, and Systems Integration (CNTRL), and Mission Planning and Execution (MiPlEx). An abstracted view of the group structure and its interaction is depicted in Figure 2.

Sensor Fusion and Environment Perception
The first step towards autonomy is self-perception and environment perception. In order to react to situations, it is important to first understand the situation. The environment perception is handled by cameras and laser sensors which are able to model a 3D obstacle mapping of the environment in real-time. This mapping can be used to build an internal map of the area from scratch and, hence, to make smart decisions concerning the mission, e.g. obstacle avoidance.

Another non-trivial part is the self-perception, i.e. the assessment of the current state of the aircraft. The self-perception comprises statements on the current position, the direction of the flight, the altitude, the flight state, the distance to

obstacles, and the acceleration. To estimate the actual state of the aircraft, the data of the already mentioned sensors are fusion-ed with the data of the global positioning system (GPS), magnetometer, and the inertial measurement unit (IMU). Therefore, to handle this bulk of data in real-time, efficient algorithms for the estimation are required. Interesting arising perception tasks are recognition of dangers, the detection and tracking of obstacles, and the automatic construction of aerial surveys.

Flight Control and Systems Integration
The actual flight of a mission is based upon commands. If the aircraft is commanded to fly on a straight line from position A to position B it would be unpleasant if the aircraft never reaches position B. In this setting, the task of flight control is to assure a smooth flight with minimal deviation. An ideal flight is quite unlikely due to environment influences, e.g. wind. Challenges are that anomaly in the behavior should be detected to enable smooth flights. Therefore, a precise flight mechanical model of the actual aircraft and robust algorithms, capable of fast adaptation to environmental conditions, are required. Additionally, it should be guaranteed that the aircraft acts inside its operational limits.

System integration is not only the task of presenting a system which is capable of flying but also of integrating all hardware as well as software components into hardware-in-the-loop simulations. Hardware-in-the-loop simulations are key to evaluate new algorithms or sensors in an early not-ready-to-flight stage of development involving a single or multiple components of the system.

Further tasks are the maintenance of the aircraft, the coordination, and the safety arrangements of the actual flight tests.

Mission Planning and Execution
The mission planning and execution represents the highest level of abstraction for commanding a UAV. This simulation is mainly achieved by encoding human-like decisions into mathematical functions either statically encoded or dynamically learned. Roughly spoken, the high aim of the group is to replace a human pilot by an auto-pilot. The auto-pilot is supposed to make all safe and reliable decisions in highly complex situations under strict time constraints.

The MiPlEx (Mission Planning and Execution) framework comprises the important aspects of achieving this decision-making. It supports planning of missions, paths, and the command execution required to achieve the goal of the mission [12]. The framework follows a layered architecture where each layer implements a different autonomy level of the system. In order to cope with the complex situations, the framework facilitates high-level behaviors, e.g. *Fly Home*, *Surveillance*, or *Search* to decrease the complexity of the planning decision-making. These high-level be-

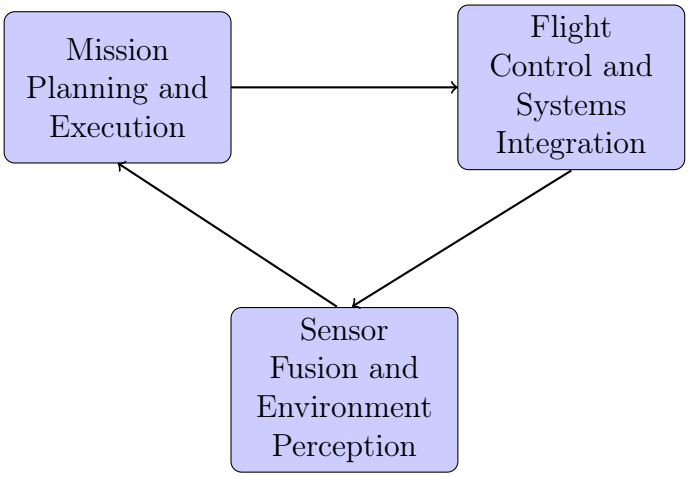


Figure 2: An abstracted overview of the group structure.

haviors are then automatically synthesized into a sequence of low-level maneuvers. Low-level maneuvers, e.g. *Take Off*, *Fly To*, or *Land*, allow the full control over the aircraft but have an high complexity for planning.

Another difficult requirement is the determination of a flight plan. Efficient flight plan computations for 3D environments are required and additionally they need to be optimized at runtime.

3.1.3 Logging

Online runtime monitoring is used to observe the data flows and to check whether they adhere to a given specification. One of the first steps in order to integrate a monitor (observe events) is to identify the interesting properties and to find the positions in the system where an easy and unharmful eavesdropping is possible. Integrate the monitor at the already existing logging positions offers several advantages. First, it simplifies the development of specifications since their development can be based on existing log files[2]. Second, both offline and online monitoring use the same data and, therefore, specifications can be reused. Last, it eases the integration of the monitor. Comparing the log files to the monitor output allows to validate the monitor.

In the following, we present some of the log files and their logged data in more detail. We abstract from the type of values and assume floating-point numbers.

[2]In most cases, specifications can be used for online and offline monitoring

- Sensor Fusion and Environment Perception:
  - Input: imu_output.log
    is used to observe the IMU data which is the most important sensor for sensor fusion and measured at 100Hz. It has nine columns: *Rotation rate* in x, y, and z axis, *acceleration* in x, y, and z axis, a *counter* to check missing values, and two time values *time_s* and *time_ms*: the first in seconds and the second in microseconds as an exact offset added to the first time value.
  - Input: mgn_output.log
    validates the IMU input on the flight attitude of the aircraft. It has five columns: *Flight attitude* in x, y, and z axis and the two time values *time_s* and *time_ms*.
  - Input: gps_vel_output.log
    is the first part of the GPS sensor, both parts are used to validate the IMU data. It contains data about the current velocity, e.g. the *horizontal* and *vertical speed*, the *actual motion direction*, as well as time values.
  - Input: gps_pos_output.log
    The second part of the GPS sensor captures information about the position. Among others, it incorporates *latitude*, *longitude*, *height*, *number of tracked observations*, and time values.
  - Output: nav_states.log
    is a Log of the output of the sensor fusion and entails 34 columns. Its main features are: current position relative to the reference point (x, y, and z), current position in World Geodetic System 1984 (WG S 84) (*latitude*, *longitude*, and *height* in meters above sea level), velocity (in x, y, and z direction), rotation rate (in x, y, and z direction), acceleration (in x, y, and z direction), *current height*, and again the two time values *time_s* and *time_ms*.
- Flight Control and System Integration - control_output.log:
  This log is generated in the control engineering part. It can contain over 100 rows and varies from flight to flight. Typical values are: the *outervel_cmd*s , representing the commands given by the mission manager, the *vel_ref* which depicts the reference behavior, and the *vel_state* which stands for the current behavior. Latter values should match with the command and differ only slightly from each other. Other values range from the *commanded velocity* or *acceleration* to the current *fuel status* or *pilot commands*. Again, *time_s* and *time_ms* are also entailed.

- Mission Planning and Execution - missionManager_output.log:
  Is the last log file we consider and it closes the circular data flow. The file contains information about the mission manager which is part of the MiPlEx and contains the information and current command of the mission. The log has 26 columns some of which are: the commanded velocity vk , the $turn\_rate$, the state of the sequence controller $stateID\_SC$, and information about the current position, e.g. x, y, and z. As before, $time\_s$ and $time\_ms$ are also entailed.

3.1.4 Current State of Software Development

This section summarizes the results presented in [44]. The development of a software project, which includes many engineers, comprises constraints on the system and the software, coding guidelines, integration, verification, and software management, i.e. tracking an issue, publishing it, assigning an engineer, and finally solving the issue.

The ARTIS development is a research project. Therefore, its constraints vary over time and are incrementally extended which introduces additional constraints on the adaptability of the software. For the management of the software a modified Mantis bug tracking server [6] is used along with an SVN [10] server for code management. The SVN server contains three repositories: green (stable version - ready for flight tests), yellow (planned version - working code which will soon be integrated into the green code), and red (development code).

In order to ease the integration of new code the overall project is composed of modules, just like the working groups. Each module selects a Module Master (MM). This MM verifies the code and then pushes the code changes into the yellow repository. Other module members do not have the permission to push changes directly into the repository. During a monthly meeting the MMs present the changes and discuss which to integrate into the green branch. The integration is planned by the Integration Master (IM) who is chosen among the MMs. She is responsible to schedule the steps to be carried out for a smooth transition. As last steps, the functionality of the code pre-version is tested to avoid negative effects and to allow a successful integration. If the IM is satisfied with the code quality this pre-version is committed into the green repository and will be used in further flight tests.

In Section 6, we receive a SVN branch of the green version in order to integrate and evaluate runtime monitoring.

3.1.5 Current State of Software Verification and Validation

In the last section, we presented the process of software integration. We often mentioned that high code quality and trustworthiness is assured by testing. In this section, we show how software verification is currently handled for sensor fusion, mission planning, and execution [44, 46].

In Section 3.1.2, the responsibility of sensor fusion is depicted. Originally, the module was written in C and featured a high quality in position estimation. It was trustworthy and, thus approved correct in principle. Unfortunately, the module had to be completely rewritten in C++ due to new software constraints, restricting the compatibility. To ensure that the new implementation is still capable of high performance, regression testing [48] is applied. In regression testing, already completed tests are continuously repeated to ensure that modifications do not alter the already tested behavior and lead to new software faults. Each test case asserts an expected value with the returned value. If the values do not differ significantly the test is passed. In this specific application, the expected values are the return values of the old C implementation which is approved correct.

In many other cases this knowledge about the expected values is missing, e.g. in mission planning and execution. They are using UML state charts, an event-based automaton, in order to implement the event handling where verification support is available. Their testing approach is broken up into several levels of abstractions each raising the confidence in the code.

First, abstract tests are used to guarantee the basic model correctness. These tests can be automatically synthesized from the model and are based upon coverage criteria of the model, e.g. transition coverage, i.e. each test run applies each transition at least once.

Second, the functionality of module components and their interaction with other module components are checked using unit tests.

Third, actual flights are simulated to check the correct integration with the other modules. Advantages of simulations are that they speed up the development life cycle, reduce the overall costs, e.g. expensive flight time, and of course increase the safety of people. Two simulations are used: software-in-the-loop (SiL) and hardware-in-the-loop (HiL).

SiL is mostly used in an earlier stage of development where hardware is not yet available or a first impression of the algorithm's performance suffices. Figure 3 shows the process. The main drawback of SiL is that the simulation (of hardware and environment) and the software is run on the same machine. This can result in a completely different runtime behavior compared to the runtime when using a dedicated embedded machine. However, SiL is flexible and no expensive hardware equipment is required.

In later development stages, hardware is available, HiL deals only with the en-

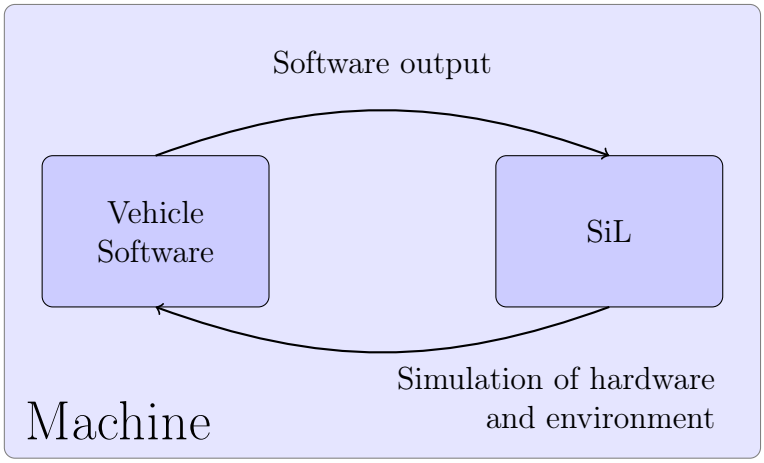


Figure 3: Illustration of the software-in-the-loop simulation setup.

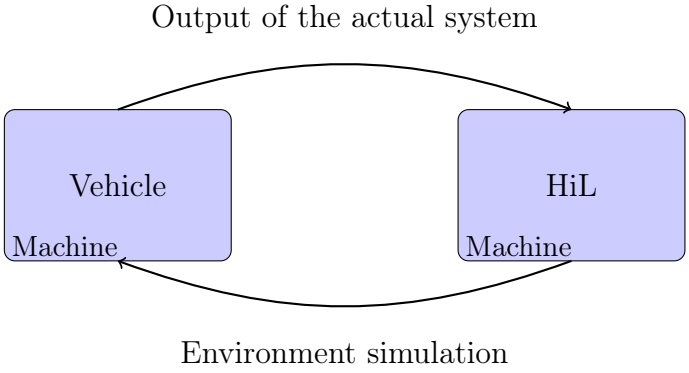


Figure 4: Illustration of the hardware-in-the-loop simulation setup.

vironment simulation where the embedded machine will be applied in the future. Figure 4 depicts the hardware-in-the-loop setup. The figure shows that the software is run on the target machine on which it will run on after deployment. The environment simulation is run on an external machine. Therefore, the behavior is closer to real flights and has more expressiveness.

Finally, the embedded system is tested during an actual flight. The obtained flight data is analyzed and shows the applicability.

3.1.6 Uncertainty about Future Regulatory Restrictions

Unmanned aircraft gain more and more public attention due to global companies like Amazon [1], Facebook [5], or Google [8]. These big corporations put a lot of effort into the development. International and national authorities are demanded to give guidelines for a legal framework and to regulate unintended operations and behavior. Frameworks like ARTIS have to keep up this fast pace in order to play an active role in creating this regulations and standards.

Currently, authorities are working on defining the categories of operations. As reference point, they often compare to available and proven regulations in manned aircraft but not all can be reused. The current regulation in Europe states that unmanned aircraft above 150 kg have to adhere EUROCONTROL (European Organisation for the Safety of Air Navigation), the others to national authorities [20, 34]. For Germany and the ARTIS framework, this implies that a human safety pilot is required who must have the aircraft in line of sight at all times in order to switch to remote control in case of an emergency [27, 33]. This is a strong and undesired intervention in the autonomy of the aircraft for any business UAV but necessary to guarantee safety of this critical domain. In order to overcome this restriction of autonomy, it is required to replace the human safety pilot by an auto-pilot which requires modeling and safety certification.

A current software standard, namely DO-178C [37], allows for the first time formal methods, object oriented techniques, and model-based development for the software certification. It distinguishes several software levels representing increasing criticality levels from *No Effect* - despite failure still safe (0 Objectives) over *Minor*(26 Obj.), *Major*(62 Obj.), and *Hazardous*(69 Obj.) up to *Catastrophic* - multiple fatalities due to failure (71 Obj.). Depending on the level, the amount of software verification effort increases, and the number of required development objectives grows. Further, the standard allows the applicability of formal methods to verify the system [38]. Open questions include how formal methods can be implemented to certify the system (preferably in an efficient, adaptive, and affordable manner) and whether a common basis of requirements of the different aircraft types can be found to reduce the effort of certification required. First steps have already been made [43, 45, 42].

This is an ongoing process and yet, the outcome, what a valid certification would require, is unclear. The research is compelled to decide whether to hold onto the simpler solution and wait how the certification situation evolves or to approach the problem right from the start by actively seeking for alternatives and maybe even propose them as a solution. The ARTIS framework has decided to evaluate the applicability of formal methods. The basic idea is to use runtime monitoring as a monitor for the overall system. Since the monitor is formally defined, the hope is to deliver it along with its proof of correctness. As we will see in Section 4.3, formal methods can guarantee bounds on the memory consumption. However, formal methods do not render the obligation to consider all possible error scenarios but they allow to *specify* such properties in a more *descriptive* way.

3.2 Applications of Runtime Monitoring

Runtime monitoring is a very powerful technique in different scenarios and stages of development. In the following, some of the possible application fields are listed. The first four applications consider more the offline monitoring aspect whereas the others indicate an incremental approach towards autonomous health management. The idea of the incremental approach is to first use monitors to ease the work of a safety pilot by pointing towards errors behaviors which he can then handle. In a second step, using these errors behaviors, the decisions of the safety pilot can be be mimicked either off-board or on-board. For now, we assume there exists a fast, secure, and strong enough data link between the system and the monitor.

Formalizing the Expectations
in the designing stage of a project, developers should be clear about what they want to achieve. Beforehand, important and expected properties can be captured in a specification. This helps early to become clear and, additionally, afterwards attained specifications can be used to check to which degree the desired properties are achieved.

Debugging, Testing, and Analysis Support
Currently, at least for the ARTIS framework, the analysis of log files is done via python plots. In case of an error, an expert has to find it by sight. He has to put the values of multiple sensors into context, make calculations, and maybe even consider the temporal dependencies of the values. Especially the last point makes this tasks very disillusioning and error-prone. A monitor can be used as a tool to narrow down the error source or even identify it. Useful techniques for this are specifying failure properties, establishing statistics, or even the filtering of the data for another analysis step on the smaller data set.

Often, it is not directly possible to specify the correct properties of a system. Either because the outcome is previously unknown, unexpected, or the bounds are not yet exactly identified. Here, statistics offer experts a deeper understanding of the system state, e.g. average trajectory deviation. Furthermore, in order to identify the bounds, monitors can evaluate *all* stored flight data and based on that give a first bound estimation.

Another scenario where monitors can be useful are tests. By extending existing unit tests with monitors, their expressiveness of a valid flight is increased and will already reduce work at an early stage of development.

Benchmarking of different Flights based on Statistics
Not only can we use (all) logged flight data to identify the bounds but also to compare different flights with each other. This is very useful when experimenting with different algorithms. It enables us to identify properties of the different algorithms on a statistical basis. For instance, one algorithm is known not to be very precise but very fast, another the contrary, precise but computational slow. Statistics offer to measure the pros and cons between those algorithms.

Integration of Blackbox Systems
Integration of new components is dangerous since it can destroy the timing behavior of the overall system. Often new components are a custom product based on a specification. This specification formalizes the properties of the delivered product, e.g. resource or timing limitations. If these properties hold the behavior of the system is expected to work in the desired way. Runtime monitoring can be used to increase the trustworthiness towards these components by supervising this specifications.

Increasing Situational Awareness of the System State
So far in, safety is guaranteed based on a human safety pilot who can switch to remote control whenever he recognizes a bad behavior. To recognize such behavior, the pilot has to observe the aircraft from the ground control station with limited information on the current internal aircraft state. This limited information and the distance between the aircraft and the pilot introduces new challenges, particularly the delayed situational awareness. Runtime monitoring can be used as an onboard reasoning mechanisms to identify unintended system states. For instance, bound violations can be passed to the pilot. These pilot notifications can have different granularities. Some violations might be safety-critical, i.e. flagged as *urgent* and some others are only simple, maybe even unnecessary notifications.

Detection of the Error Source
Runtime monitoring is a suitable technique for *identifying* system faults. The next step towards autonomous health management will be to diagnose the faults to find its source. In order to illustrate this point, assume we have three sensory inputs for a system property a. If two of the sensors say a holds and the other says it does not then it is very likely that a holds and the differently reporting sensor is erroneous. This is just an example to show what diagnosis might look like. Diagnosis reasoning can be arbitrary high, e.g. the trustworthiness of the sensors in different situations can be taken into account. That diagnosis is also very useful for a human safety pilot at the ground control station.

Integration of a Contingency Manager
To go from these open-loop use cases to a closed health managing loop, the outcome of the diagnoses can be passed to a contingency manager. Depending on the design choice, this can either be part of the monitor or a separate component. In both cases, the task of the contingency manager will be to mitigate the error impact towards the system. Countermeasures can be initiated which were originally the task of the human safety pilot. However, in some cases, e.g. fatal error, the mitigation of the failure might not be possible and a safe flight behavior cannot be re-established. Yet, in this worst case scenario, a possibility might be to implement actions which guarantee at least a controlled form of action, safe landing, or flight termination.

4 The Specification Language Lola

Lola is a stream-based specification language for synchronous systems, first introduced in [22]. The general idea of Lola is to generate a set of *output* streams, based on a given set of *input* streams. Input streams describe the values of the system under observation and output streams represent errors or diagnosis reports of the monitor. In this section, we give an overview on the essential definitions introduced in this paper.

Synchronous systems are systems in which the variables change their values by a unique and synchronized clock signal. In Lola, the system variables are represented as streams. A specification describes the dependencies between *input* streams and *output* streams. A stream has previous values, a present value, and possibly future values. The dependencies express the temporal connection between all streams. In fact in Section 4.3, we will see that LTL [35] properties are expressible in Lola.

Further, Lola streams are typed and the language allows to specify *incrementally computable statistical measures* which enrich the information content for a user. Additionally, the same Lola specification can be used for online and offline monitoring. Despite its expressiveness, the language is kept simple and closes the gap between temporal logic and hand-written monitor code.

In this section, the basic structure of a Lola specification is introduced and the underlying monitoring algorithm is presented. In Section 4.3 the known efficiently monitorable fragment of Lola is shown. To ease the introduction, the left open syntax is restricted to the one used for the first Lola implementation.

4.1 Syntax & Semantics

Lola specifications are structured in a modular way, depicted in Syntax 1. Writing specifications in a modular way is a key feature of Lola since it helps a specification engineer to keep track of the current state of the target property.

4.1.1 Syntax

The specification consists of input, constant, and output streams each having a unique identifier and a type. The basic types are integer and boolean. Syntax 2 shows the possible values of the types and the identifiers. The syntax is given in Extended Backus-Naur Form (EBNF). EBNF consist of terminal symbols (indicated by '..') and non-terminal production rules (indicated by ⟨..⟩). Further, production rules can contain optional symbols (indicated by [..]), grouping symbols (indicated by (..)), repetition symbols (indicated by *) and alternation symbols (indicated by |). The rules are used to express how the terminal symbols are allowed to

be combined. Input streams represent the values given to the monitor, i.e. the values under test. Constant streams are streams with a fixed value. Especially in specifications, where fixed bounds often reoccur, constants allow adjusting the specification by only changing these single points.

Last, output streams are used to handle the error checks (*true* or *false*) and the statistical measurement computations by specifying expressions. Further, a boolean output stream can be declared as *trigger* which generates a notification to the user each time the value of the output evaluates to true. Syntax 3 lists the possible *expressions* for the output. The basic integer and boolean operators are available and an if-statement operator which is similar to common programming languages.

⟨*lola-format*⟩ ::= ϵ | ⟨*streamDef*⟩ ⟨*lola-format*⟩

⟨*streamDef*⟩ ::= ⟨*inputDef*⟩
| ⟨*constantDef*⟩
| ⟨*outputDef*⟩
| ⟨*triggerDef*⟩

⟨*inputDef*⟩ ::= '`input`' ⟨*type*⟩ ⟨*identifier*⟩

⟨*constantDef*⟩ ::= '`const`' ⟨*type*⟩ ⟨*identifier*⟩ '`:=`' ⟨*literal*⟩

⟨*outputDef*⟩ ::= '`output`' ⟨*type*⟩ ⟨*identifier*⟩ '`:=`' ⟨*expression*⟩

⟨*triggerDef*⟩ ::= '`trigger`' ⟨*identifier*⟩

Syntax 1: Structure

⟨*type*⟩ ::= ‘`int`’ | ‘`bool`’

⟨*identifier*⟩ ::= (‘`a`’-‘`z`’|‘`A`’-‘`Z`’) (‘`a`’-‘`z`’|‘`A`’-‘`Z`’ | ‘`0`’-‘`9`’ | ‘`_`’)*

⟨*literal*⟩ ::= ‘`true`’ | ‘`false`’ | ⟨*integer*⟩

⟨*integer*⟩ ::= [-] (‘`0`’-‘`9`’) (‘`0`’-‘`9`’)*

Syntax 2: Literals

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩ | ⟨*unaryExpr*⟩
| ⟨*binaryExpr*⟩ | ⟨*ifExpr*⟩ | ⟨*shiftExpr*⟩
| ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*unaryExpr*⟩ ::= ‘`!`’ ⟨*expression*⟩

⟨*binaryExpr*⟩ ::= ⟨*expression*⟩ ⟨*comparison*⟩ ⟨*expression*⟩
| ⟨*expression*⟩ ⟨*computation*⟩ ⟨*expression*⟩

⟨*comparison*⟩ ::= ‘`<`’ | ‘`<=`’ | ‘`=`’ | ‘`!=`’ | ‘`>=`’ | ‘`>`’

⟨*computation*⟩ ::= ‘`&`’ | ‘`|`’ | ‘`->`’ | ‘`<-`’ | ‘`<->`’
| ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`^`’

⟨*ifExpr*⟩ ::= ‘`if`’ ⟨*expression*⟩ ‘`{`’ ⟨*expression*⟩ ‘`}`’ ⟨*elseExpr*⟩

⟨*elseExpr*⟩ ::= **[** ‘`elif`’ ⟨*expression*⟩ ‘`{`’ ⟨*expression*⟩ ‘`}`’ ⟨*elseExpr*⟩ **]**
‘`else`’ ‘`{`’ ⟨*expression*⟩ ‘`}`’

⟨*shiftExpr*⟩ ::= ⟨*identifier*⟩ ‘`[`’ ⟨*integer*⟩ ‘`,`’ (⟨*literal*⟩ | ⟨*identifier*⟩) ‘`]`’

Syntax 3: Expressions

Definition 1. Well-Typed Specifications
LOLA is a typed language. The syntax of a specification is restricted based on type conventions. Let T be the set of types consisting of *int* and *bool* and S be the set containing all streams. Each specified input, constant, or output stream $s_i \in S$ has a certain type $t_i \in T$, where $0 < i \leq \mid S \mid$. A specification is called *well-typed* if all output streams s_i and their corresponding *expressions* e_i satisfy the following type rules, where e_j, e_k, and e_m indicate subexpressions.

- If s_i is of type t_i then e_i has to be of type t_i as well.
- If e_i is a *literal* of type t_j then e_i is of type t_j.
- If e_i represents an *identifier* s_j then e_i is of type t_j.
- If $e_i \mathrel{\widehat{=}} ! \, e_j$ then the type of e_j and e_i have to be of type *bool*.
- If $e_i \mathrel{\widehat{=}} e_j \circ e_k$ then
  - If $\circ \in \{\&, |, \leftarrow, \rightarrow, \leftrightarrow\}$ then both e_j and e_k have to be of type *bool* and e_i is of type *bool*.
  - If $\circ \in \{+, -, *, \backslash, \%, \hat{}\}$ then both e_j and e_k have to be of type *int* and e_i is of type *int*.
  - If $\circ \in \{<, \leq, =, \neq, \geq, >\}$ then both e_j and e_k have to be of type *int* and e_i is of type *bool*.
- If $e_i \mathrel{\widehat{=}} \; if \; e_j \; \{e_k\} \; else \; \{e_m\}$ then e_j has to be of type *bool*, e_k and e_m have to be of the same type t_k and e_i is of type t_k.
- If $e_i \mathrel{\widehat{=}} e_j[n, \; l]$ then e_i and e_j have to be of the same type t_j and further l has to be either a *literal* or *constant* of type t_j. n is a natural number. In the following, we call this operator the *offset operator*.

Note that $e \mathrel{\widehat{=}} \; if \; e_1 \; \{e_2\} \; elif \; e_3 \; \{e_4\} \; else \; \{e_5\}$ is syntactic sugar for
$e \mathrel{\widehat{=}} \; if \; e_1 \; \{e_2\} \; else \; \{if \; e_3 \; \{e_4\} \; else \; \{e_5\}\}$. In the following, we assume well-typed specification unless explicitly stated differently.

Example 1. Bound Check
In order to prepare the more complex formal semantics, a simplistic boundary check is shown in Listing 1. The well-typed specification is used to check whether a given input *value* exceeds a predefined constant *bound*. In case the check is *enabled* and the input exceeds the bound, a notification is raised. The specification can be understood as an equation system. The set of variables consists of the two inputs *enabled*, *value*, and the output stream *exceeds*. Note that a monitor evaluates a

continuous trace of the system. The evaluation of the LOLA specification starts at the first position (indexing starting with zero) and traverses the trace onwards. Therefore, at each position of the trace an instance of this equation system is spawned and possibly resolved. Simplifying the argumentation, we introduce the shorthand *<identifier>#<integer>* which is an abbreviation for *<identifier> at position <integer> of the trace*. Assume *enabled#1* is *true* and *value#1* is *5* at the second position in the trace. In this case, *exceeds#1* corresponds to *if true {5 > 10}else{false}* which can be further resolved to $false$. Since the result is *false* no trigger is raised. If the next input values *enabled#2* and *value#2* would be *true* and *100* then the output stream instance would evaluate to *true* which would raise a trigger notification.

```
1 input    bool enabled
2 input    int  value
3 const    int  bound   := 10
4 output   bool exceeds := if enabled { value > bound } else { false }
5 trigger exceeds
```

Listing 1: A simple LOLA specification that checks whether an input *value* exceeds a predefined *bound* and the *enabled* holds.

4.1.2 Semantics

The example above gave a first intuition on how a specification is evaluated over an input trace $\tau \mathrel{\widehat{=}} \langle \tau_1, \ldots, \tau_m \rangle$, representing literal values for each input stream $s_1^{in}, \ldots, s_m^{in}$. In the following, the relation between input streams and output streams is formally defined. Let N be the length of τ and L be the set containing all literals. Further, let $s_1^{out}, \ldots, s_n^{out}$ be the specified output streams with their corresponding expressions $e_1, \ldots, e_n$ and $s_1^{const}, \ldots, s_k^{const}$ be the given constant streams representing the constant values $c_1 \in L, \ldots, c_k \in L$. Then, the evaluation over the trace is defined as a stream of N tuples $\langle \sigma_1, \ldots, \sigma_n \rangle$ where for all $0 \leq j < N$ and $0 < i \leq n$, the following equation has to be satisfied:

$$\sigma_i(j) = eval(e_i)(j)$$

Such an equation satisfying evaluation is called an *evaluation model*. $\sigma_i(j)$ stands for the value access of the output stream evaluation σ_i at position j. Similar, $\tau_i(j)$ is used to access position j of the input trace τ_i.

eval is a function which, given an expression and a position, returns the result of the expression. It is inductively defined as follows:

- Base case:
  - $eval(l)(j) = l$, where $l \in L$
  - $eval(s_i^{const})(j) = c_i$, where $0 < i \leq k$
  - $eval(s_i^{in})(j) = \tau_i(j)$, where $0 < i \leq m$
  - $eval(s_i^{out})(j) = \sigma_i(j)$, where $0 < i \leq n$
- Inductive case, where e_l, e_r, and e_c are sub-expressions:
  - $eval(\ !\ e_c)(j) = !\ eval(e_c)(j)$
  - $eval(e_l \circ e_r)(j) = eval(e_l)(j) \circ eval(e_r)(j)$, where $\circ$ is any binary operator
  - $eval(\ (\ e_c\ )\ )(j) = (\ eval(e_c)(j)\ )$
  - $eval(e_l\ \circ_1\ e_c\ \circ_2\ e_r)(j) \begin{cases} (eval(e_l \circ_1 e_c)(j))\ \circ_2\ eval(e_r)(j) & \text{, if } left(\circ_1, \circ_2) \\ eval(e_l)(j)\ \circ_1\ (eval(e_c \circ_2 e_r)(j)) & \text{,otherwise.} \end{cases}$
    where $left(\circ_1, \circ_2)$ evaluates to $true$ when the precedence order, given in Figure 5, of $\circ_1$ is higher or the operator is left-associative and has an equal order, otherwise it evaluates to $false$.
  - $eval(\text{if } e_c\ \{e_l\} \text{ else } \{e_r\})(j) = \text{if } eval(e_c)(j)\ \{eval(e_l)(j)\} \text{ else } \{eval(e_r)(j)\}$
  - $eval(e[p, oobv])(j) = \begin{cases} eval(e)(j + eval(p)(j)) & \text{, if } 0 \leq j + eval(p)(j) < N \\ eval(oobv)(j) & \text{, otherwise.} \end{cases}$
    where $p \in \mathbb{N}$ and *oobv* represents the out-of-bounds value $\in L$. This operator will be further called *offset operator*.

In the previous example, the given specification consists only of dependencies on present stream values. In the semantics above, the so called *offset operator* is already formally introduced which gives access to past and future values of an existing stream. Accessing these temporal values allows the computation of statistical measures.

Example 2. First Statistical Measure
In Listing 2, we compute the sum over the input *value* in two ways using the offset operator. The output stream *sum_backward* computes the sum by accessing the previous value, i.e. [**-1**, 0], whereas *sum_forward* computes the sum by accessing the future value, i.e. [**1**, 0]. Let $\langle 1, 2, 3 \rangle$ be the input trace for the input *value* stream, we evaluate the outputs in an online manner as follows:

- *sum_backward*:
  1. $sum\_backward\#0 = value\#0 + sum\_backward\#\text{-}1 = 1 + 0 = 1$
  2. $sum\_backward\#1 = value\#1 + sum\_backward\#0 = 2 + 1 = 3$
  3. $sum\_backward\#2 = value\#2 + sum\_backward\#1 = 3 + 3 = 6$
- *sum_forward*:
  1. $sum\_forward\#0 = value\#0 + sum\_forward\#1 = 1 + ? = ?$
  2. $sum\_forward\#1 = value\#1 + sum\_forward\#2 = 2 + ? = ?$
  3. $sum\_forward\#2 = value\#2 + sum\_forward\#3 = 3 + 0 = 3$
     $sum\_forward\#1 = 2 + sum\_forward\#2 = 2 + 3 = 5$
     $sum\_forward\#0 = 1 + sum\_forward\#1 = 1 + 5 = 6$

The stream *sum_backwards* inserts the out-of-bounds value at the start of the trace. Since we try to access a previous value which can never exist. *sum_forwards* works similar but inserts the value at the end of the trace. Being at the end guarantees that no further values exist. In Section 4.3, we will see that the former is more desirable since back propagation is not required. Note that we computed the sum over the values correctly but the position holding the result differs.

```
1 input    int value
2 output   int sum_backward   :=  value + sum_backward[-1, 0]
3 output   int sum_forward    :=  value + sum_forward [ 1, 0]
```

Listing 2: A simple Lola specification which computes the sum of a given input stream in a backward manner and in a forward manner.

4.1.3 Well-Formedness

In Section 4.1.1, the syntax was restricted to guarantee well-typed specifications. In this section, another restriction is imposed on the specification in order to have a unique evaluation.

Definition 2. Well-Defined Specifications [22]
A well-typed specification is also called *well-defined* if the evaluation model is unique for arbitrary input streams of equal length.

Example 3. Well-Definedness
Consider Listing 3, the input *value* with assumed length N and five outputs are declared. For this example, we consider each combination of the input with one output to be an independent specification since well-definedness is a property of the specification as a whole. We will use $0 \leq j < N$ as an arbitrary but valid position of the trace.

- *stream_1* is well-defined. There is exactly *one* evaluation model which is given by: $\sigma_{stream\_1}(j) = true \leftrightarrow \tau_{value}(j) < 10$.

- *self* is ill-defined. There is *no* evaluation model. We can assign *self* to $true$ or $false$ but both result in a contradiction.

- *cyclic_1* and *cyclic_2* are both ill-defined. For both of them, there is *more than one* evaluation model. $\sigma_{cyclic\_1}(j) = true$ and $\sigma_{cyclic\_2}(j) = true$; $\sigma_{cyclic\_1}(j) = false$ and $\sigma_{cyclic\_2}(j) = false$.

- *stream_2* is well-defined. $(value < 10$ & $value >= 10)$ always evaluates to $false$ which leads to the *unique* evaluation model: $\sigma_{stream\_2}(j) = true$.

Well-definedness is a powerful semantical restriction on the specification. But, in order to identify an ill-defined specification it is essential to enumerate *all* possible evaluation models for *all* possible input traces which is very expensive to check. For instance in Listing 3, we saw that *stream_2* is well-defined but at a first glance one might guess that it is ill-defined due to the self reference.

left-associative	right-associative
$*,\ \backslash,\ \%$	$\hat{}$
$+,\ -$	
$\&$	
$\mid$	
$\leftarrow,\ \rightarrow$	
$\leftrightarrow$	
$<,\ \leq,\ =,\ \neq,\ \geq,\ >$	

Figure 5: Operator Precedence, highest at the top.

```
1 input   int   value
2 output  bool  stream_1  :=  value < 10
3 output  bool  self      :=  ! self
4 output  bool  cyclic_1  :=  cyclic_2
5 output  bool  cyclic_2  :=  cyclic_1
6 output  bool  stream_2  :=  !( value < 10 & value >= 10 ) | ! stream_2
```

Listing 3: If we split the specification into five, for each input/output stream pair then the specifications enumerated by the lines are: well-defined (line: 2, 6) and ill-defined (line: 3, 4, 5).

To avoid this expensive check, a syntactical restriction is introduced on the specification called *well-formedness*. Well-formedness analyzes the so called *dependency graph* which can be done comparatively cheap. In [22], the authors proved that *well-formedness* implies *well-definedness*.

Definition 3. Dependency Graph [22]
A *dependency graph* for a given Lola specification is a directed and weighted multi-graph $G = \langle V, E \rangle$. The task of the graph is to capture the temporal dependencies between the streams. Therefore, the vertex set V consists of all the input and output streams $\{s_1^{in}, \ldots, s_m^{in}, s_1^{out}, \ldots, s_n^{out}\}$, and their temporal dependencies are encoded using weights on the edges E. An edge $e : \langle s_i^{out}, s_k^{out}, w \rangle$ from vertex s_i^{out} to s_k^{out} with weight w, respectively $e : \langle s_i^{out}, s_k^{in}, w \rangle$, is contained in the edge set E iff for some position j, $\sigma_i(j)$ contains a subexpression $\sigma_k(j+w)$, respectively $\tau_k(j+w)$.

Note that the graph needs to be a multi-graph to represent that an expression can have several temporal subexpressions with different temporal dependencies to the same stream. Further, an input stream cannot have outgoing edges since they are independent. After introducing the notion of a dependency graph, it is possible to define how we can traverse it. A *walk* is defined on the graph and properties of it. With these properties *well-formedness* is imposed on Lola specifications. As a reminder, *well-formedness* implies *well-definedness* but is a specification check on the syntax instead of its semantics.

Definition 4. Walk [22]
A *walk* on a dependency graph $G = \langle V, E \rangle$ is a sequence $v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots v_{k-1} \xrightarrow{e_{k-1}} v_k$ such that for all $k > 0$, $v_k \in V$ and $e_k : \langle v_k, v_{k+1}, w \rangle \in E$. A *walk* is called closed whenever $v_1 = v_k$. The total weight of a walk is the sum of all weights along the walk.

Definition 5. Well-Formed Specifications [22]
A well-typed specification is also called *well-formed* if there exists no closed walk with total weight zero.

Example 4. Average Computation
In Listing 4, we compute the average value of a given input stream. Additionally, we are interested whether the average keeps increasing along the trace. We use four output streams for the computation and a trigger condition which notifies us when we are decreasing. The first stream is called *sum* and is identical to the already seen *sum_ backward* from Listing 2. In order to capture the current position in the trace, i.e. the amount of already given values, we use the output stream *pos*. The output stream *avg* computes the average by dividing the *sum* by the current *pos*ition. Since we use integer streams, the value is floored. Finally with *dec*, we are able to identify whether our average decreased, which would trigger a notification, by comparing the previous *avg* value with the current *avg* value. The corresponding dependency graph is depicted in Figure 6. The stream *dec* shows why we require the graph to be a multi-graph. The only closed walks in the graph are the self-loops on *sum* and *pos*. In both cases, their total weight is non-zero from which we can conclude that the specification is *well-formed* and, thus, also *well-defined*.

```
1 input    int    value
2 output   int    sum    :=  sum[-1 , 0] + value
3 output   int    pos    :=  pos[-1, 0] + 1
4 output   int    avg    :=  sum / pos
5 output   bool   dec    :=  avg < avg[-1,0]
6 trigger dec
```

Listing 4: A LOLA specification that computes the average value along the trace.

In [22], it was shown that well-definedness does not always imply well-formedness. The given counterexample is *output bool s := s && !s* which is well-defined, i.e. has a unique evaluation model ($\sigma_s(j) = false$, for any position j) but is not well-formed, i.e. there exists a closed walk with total weight 0 (zero self-loop). The same holds for Line 6 in Listing 3.

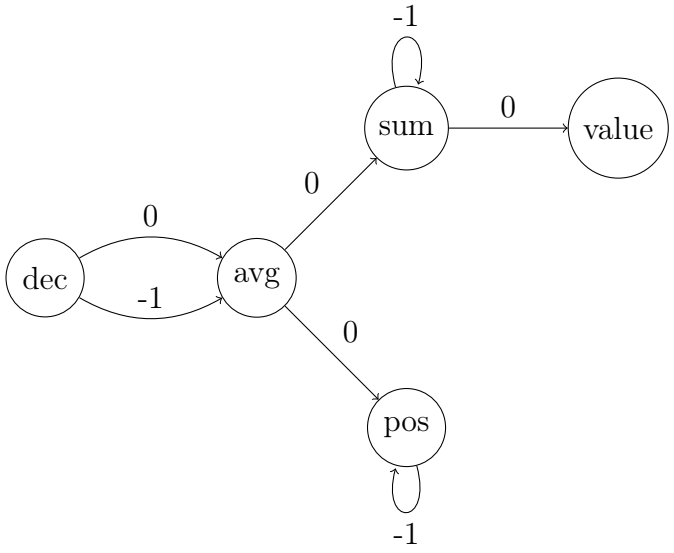


Figure 6: Dependency graph for the specification given in Listing 4.

4.2 Monitoring Algorithm

In this subsection, the algorithm for Lola *online monitoring* is presented. Since the only difference to *offline monitoring* is the access to the completed trace, the same algorithm can be applied for *offline monitoring*.

Let φ be a Lola specification where $S^{in} = \{s_1^{in}, \ldots, s_m^{in}\}$ is the set of inputs, $S^{out} = \{s_1^{out}, \ldots, s_n^{out}\}$ is the set of outputs, and $S^{const} = \{s_1^{const}, \ldots, s_k^{const}\}$ is the set of constants. Further, let L be the set of all possible literals and $\tau = \langle \tau_1, \ldots \tau_m \rangle$ be an input trace of length N for each defined input stream.

The *Evaluation Algorithm* is given in Algorithm 1. It is based upon two sets of equations, called stores: the *resolved store* R and the *unresolved store* U. All resolved equations will be contained in R and all unresolved equations will be in U. An equation is called resolved when it is of the following form.

- $\tau_k(j) = l$, where $0 < k \leq m$, $0 \leq j < N$, and $l \in L$
- $\sigma_k(j) = l$, where $0 < k \leq n$, $0 \leq j < N$, and $l \in L$

Otherwise the equation is unresolved.

First, all occurrences of constants are replaced by their literal values. Second, both stores are initialized and the current position in the trace is set. Then, in Line 6 and 9 it is computed when it is safe to remove resolved equations, i.e. when their value is not required anymore. This computation is similar for both input streams and output streams and is based on the maximum lookup of previous values. For instance, in Listing 4, rem_{value}^{in} would evaluate to 0 which indicates that in the

future this value is not required anymore and, therefore, can safely be removed in the present. In contrast, rem^{out}_{sum} would return 1 which means that this value has to be stored for one trace step and, only then, can be removed safely.

The equation evaluation starts in line 12. The upcoming equations are incrementally managed while traversing the trace. In each synchronous step, new input stream values are added to R and new output stream equations are spawned and added to U. In Line 20, the equations in U are simplified as much as possible and, if they turn out to be resolved, they are removed from U and added to R. If declared, they can trigger a notification. *Equation eq is simplifiable*, respectively *Simplify(eq)* means that at least one of the following rules applies:

- Partial evaluation: $true \wedge e \rightsquigarrow e$, $e + 0 \rightsquigarrow e$, and such ...
- Rewriting rules: if $true\ \{e_1\}\ else\ \{e_2\} \rightsquigarrow e_1$, and such ...
- Substitution: if $\sigma_k(j)$ respectively $\tau_k(j)$ occurs in eq and $\sigma_k(j) = l \in R$ respectively $\tau_k(j) = l \in R$ then it is substituted with the occurrence l.

Before starting the next equation evaluation round, the unrequired equations are removed from R using the previously computed *rem* values.

Example 5. Evaluation Algorithm Example
To conclude this subsection, the algorithm is illustrated by means of an example shown in Listing 5. We have one input stream *signal* with a value range from 0 to 10 and two output streams *dec* and *count*. The stream *dec* checks the *signal* whether the value will be decreasing and *count* captures how often this was the case. Let the input trace τ_{signal} be given by $\langle 1, 3, 2, 5, 4 \rangle$. Table 1 illustrates the incremental approach of the algorithm, starting with position $i = 0$ of the trace. Notice that we abbreviate the equations in R and U by their left hand side. At the first position, we read the input value 1. The current equation of *dec* and *count* are added to the unresolved store U. Since we are at the first position in the trace, we have to insert the out-of-bounds values for the equations which contain past dependencies. Thereafter, we simplify the unresolved equations in U. Since we do not have any resolved equations, there is nothing to add to the resolved store R. Therefore, we remove the resolved equations which are irrelevant for the further evaluation and continue at the next position. At the second position, we add the new resolved and unresolved equations and try to simplify existing elements of U, as before. This time, having $\tau_{signal}(1)$, we can simplify $\sigma_{dec}(0)$ to $false$ which further results in the simplification of $\sigma_{count}(0)$ to 0 and, then, in the substitution of $\sigma_{count}(0)$ by 0 in $\sigma_{count}(1)$. At the end of this iteration, we have to remove three entries from R: $\tau_{signal}(1) = 3, \sigma_{dec}(0) = false$, and $\sigma_{count}(0) = 0$. The further evaluation steps work analogously, except that at the last position, we have to insert the out-of-bounds values for the equations which are referring to future values, as those cannot exist. Notice that the final *count* value is 2 which captures exactly the decrease from position 1 to 2 and 3 to 4.

```
1 input   int   signal
2 output  bool  dec    := signal > signal[1, 10]
3 output  int   count  := count[-1, 0] + if dec {1} else {0}
```

Listing 5: A LOLA specification that computes how often the input *signal* is frozen.

Regarding both time and space, the algorithm is linear in the length of the trace and the size of the specification [22]. Especially for online monitoring, where the length of the trace is a-priori unknown, this imposes restrictions on the applicability. Therefore, in the next subsection, a class of LOLA specifications is presented which can be efficiently monitored due to a constant bound on the number of equations in U.

i	$\tau_{signal}(i)$	$\sigma_{dec}(i)$	$\sigma_{count}(i)$
0	1	$1 > \tau_{signal}(1)$	$0 +$ if $\sigma_{dec}(0)$ $\{1\}else\{0\}$

$R = \{\tau_{signal}(0)\}$
$U = \{\sigma_{dec}(0),\ \sigma_{count}(0)\}$
Simplifiable: $\sigma_{count}(0) =$ if $\sigma_{dec}(0)$ $\{1\}else\{0\}$
$R = \{\tau_{signal}(0)\}$
$U = \{\sigma_{dec}(0),\ \sigma_{count}(0)\}$
Removable: $\tau_{signal}(0)$

1	3	$3 > \tau_{signal}(2)$	$\sigma_{count}(0) +$ if $\sigma_{dec}(1)$ $\{1\}else\{0\}$

$R = \{\tau_{signal}(1)\}$
$U = \{\sigma_{dec}(0),\ \sigma_{count}(0), \sigma_{dec}(1),\ \sigma_{count}(1)\}$
Simplifiable: $\sigma_{dec}(0) = 1 > 3 = false$, $\sigma_{count}(0) = 0$,
$\sigma_{count}(1) = 0 +$ if $\sigma_{dec}(1)$ $\{1\}else\{0\}$
$R = \{\tau_{signal}(1), \sigma_{dec}(0), \sigma_{count}(0)\}$
$U = \{\sigma_{dec}(1),\ \sigma_{count}(1)\}$
Removable: $\tau_{signal}(1), \sigma_{dec}(0), \sigma_{count}(0)$

2	2	$2 > \tau_{signal}(3)$	$\sigma_{count}(1) +$ if $\sigma_{dec}(2)$ $\{1\}else\{0\}$

$R = \{\tau_{signal}(2)\}$
$U = \{\sigma_{dec}(1),\ \sigma_{count}(1), \sigma_{dec}(2),\ \sigma_{count}(2)\}$
Simplifiable: $\sigma_{dec}(1) = 3 > 2 = true$, $\sigma_{count}(1) = 1$,
$\sigma_{count}(2) = 1 +$ if $\sigma_{dec}(2)$ $\{1\}else\{0\}$
$R = \{\tau_{signal}(2), \sigma_{dec}(1), \sigma_{count}(1)\}$
$U = \{\sigma_{dec}(2),\ \sigma_{count}(2)\}$
Removable: $\tau_{signal}(2), \sigma_{dec}(1), \sigma_{count}(1)$

3	5	$5 > \tau_{signal}(4)$	$\sigma_{count}(2) +$ if $\sigma_{dec}(3)$ $\{1\}else\{0\}$

$R = \{\tau_{signal}(3)\}$
$U = \{\sigma_{dec}(2),\ \sigma_{count}(2), \sigma_{dec}(3),\ \sigma_{count}(3)\}$
Simplifiable: $\sigma_{dec}(2) = 2 > 5 = false$, $\sigma_{count}(2) = 1$,
$\sigma_{count}(3) = 1 +$ if $\sigma_{dec}(3)$ $\{1\}else\{0\}$
$R = \{\tau_{signal}(3), \sigma_{dec}(2), \sigma_{count}(2)\}$
$U = \{\sigma_{dec}(3),\ \sigma_{count}(3)\}$
Removable: $\tau_{signal}(3), \sigma_{dec}(2), \sigma_{count}(2)$

4	4	$4 > 10 = false$	$\sigma_{count}(3) + 0$

$R = \{\tau_{signal}(4)\}$
$U = \{\sigma_{dec}(3),\ \sigma_{count}(3), \sigma_{dec}(4),\ \sigma_{count}(4)\}$
Simplifiable: $\sigma_{dec}(3) = 5 > 4 = true$, $\sigma_{count}(3) = 2$,
$\sigma_{count}(4) = 2 + 0 = 2$
$R = \{\tau_{signal}(4), \sigma_{dec}(3), \sigma_{count}(3), sigma_{dec}(4),\ \sigma_{count}(4)\}$
$U = \{\ \}$

Table 1: An algorithmic LOLA evaluation based on Example 1.

Algorithm 1 Lola Evaluation Algorithm

Input: Specification $\varphi = \langle S^{in}, S^{out}, S^{const} \rangle$ with literal set L,
$\tau = \langle \tau_1, \dots \tau_m \rangle$ with length N

Substitute all occurrences of $c \in S^{const}$ in all $s \in S^{out}$.

$i \leftarrow 0$ ▷ Current position in the trace.
$R \leftarrow \{\}$
$U \leftarrow \{\}$

for all $s_k \in S^{out}$ **do** ▷ When to remove resolved output equations.
    $rem_k^{out} \leftarrow max(\{o \mid o \geq 0 \text{ and } s_k[-o, d] \text{ is a subexpression} \in \varphi\} \cup \{0\})$
end for

for all $s_k \in S^{in}$ **do** ▷ When to remove resolved input equations.
    $rem_k^{in} \leftarrow max(\{o \mid o \geq 0 \text{ and } s_k[-o, d] \text{ is a subexpression} \in \varphi\} \cup \{0\})$
end for

while $i < N$ **do**
    wait until $\tau(i)$ is available. ▷ Synchronous system step.

    for all inputs $s_k^{in} \in S^{in}$ **do** ▷ Insert new input equations.
        add $\tau_k(i) = l$ to R.
    end for

    for all outputs $s_k^{out} \in S^{out}$ **do** ▷ Insert new output equations.
        add $\sigma_k(i) = eval(e_k)(i)$ to U.
    end for

    while $\exists$ equation eq: $\sigma_k(j) = eval(e_k)(j) \in U$ which is simplifiable **do**
        Simplify(eq) ▷ Apply: partial evaluation, rewriting, and substitution.
        if resolved(eq) **then** ▷ Resolved if eq: $\tau_k(j) = l$ or $\sigma_k(j) = l$, $l \in L$.
            remove eq from U.
            add eq to R.
            if marked as trigger **then**
                notify()
            end if
        end if
    end while

    for all equations $eq \in R$ **do** ▷ Remove what is not needed anymore.
        if eq: $\sigma_k(j) = l$ and $j + rem_k^{out} \leq i$ **then**
            remove eq from R
        end if
        if eq: $\tau_k(j) = l$ and $j + rem_k^{in} \leq i$ **then**
            remove eq from R
        end if
    end for

    $i \leftarrow i + 1$
end while

4.3 Efficiently Monitorable Fragment

Consider Listing 6. In this example, some popular linear temporal logic (LTL) operators are encoded into a LOLA specification. The quantifier *globally* means that a property, here a, should hold along the whole trace. The quantifier *eventually* represents the obligation that the property, here b, has to hold at least once along the trace. The last operator, the *until* operator, assures that property a holds until property b holds. Notice that the given out-of-bounds values determine how unresolved obligations are resolved at the end of the trace. For instance, currently at the end of the trace, *until* requires b to hold once to evaluate to $true$ which would not be the case if $true$ is used as out-of-bounds value.

In general, the satisfaction of an LTL formula is based on the initial state whereas LOLA streams can successively compute intermediate results. For instance, assuming the input trace $\langle false, true, true \rangle$, the LTL operator globally would return $false$ since the first position does not hold. Whereas in LOLA, *globally* would return an ambiguous output evaluation stream $\langle false, true, true \rangle$. As an alternative, to facilitate only a single *notification* if the formula holds, and none if it does not, one could define a trigger on an additional stream. This stream requires two obligations to hold: first, the LTL operator should evaluate to $true$ and second, we need to be at the first position. For instance for *eventually*, we could use a stream like:
`output bool trigger_E := (a[-1, true] & !a[-1, true]) & eventually`
In the following, we refrain from this difference.

```
1 input bool a
2 input bool b
3 output bool globally    :=  a & globally[1, true]
4 output bool eventually  :=  b | eventually[1, false]
5 output bool until       :=  b | (a & until[1, false])
```

Listing 6: A LOLA specification which encodes some of the LTL operators.

In the worst case, all three output streams cannot be monitored efficiently. Assume the trace τ_a contains only $true$ values, then the first *globally* equation can be evaluated at the end of the trace at the earliest. The reason for this is that a partial evaluation is not possible. Only at the end of the trace, we can use the out-of-bounds value which results in a backward propagation of the resolved equations. Analogously for *eventually*, the same holds if only $false$ values are contained in the trace τ_b. Next, the notion of *efficiently monitorable* specifications is formally defined and a syntactical characterization is given.

Definition 6. Efficiently Monitorable Specification
A LOLA specification is *efficiently monitorable*(EM) if the memory consumption needed to monitor it is constant in the size of the trace, i.e. there exists a constant bound for the worst case.

Essentially, efficiently monitorable specifications refer only to past or boundable future stream evaluations, i.e. streams for which there exists a point in time where we can predict its evaluate, e.g. future input values.

Definition 7. Syntactical Characterization of an EM Specification
Given a LOLA specification φ and its dependency graph G. φ is called efficiently monitorable if G has no closed walk with a positive total weight.

Here, the proof is omitted that the characterization suffices to bound the memory consumption to a constant in the length of the trace. The proof idea is to construct a function for each node in G which computes the maximal reference, i.e. offset, into the future. This maximum cannot be infinite because the graph is finite and there exists no closed walk.

As mentioned, the examples in Listing 6 are all *non*-efficiently monitorable and the characterization covers this due the positive self-loops in its dependency graph. Fortunately, it is often possible to reformulate *non-efficiently monitorable* specifications into *efficiently monitorable* specifications.

For example, `output bool always := a & always[-1, true]` is perhaps more natural and could be used instead of *globally*. But similar to Listing 2 the position of the final value changes. Similar, instead of *eventually* one could use `output bool finally := b | finally[-1, true]` to capture that the obligation is already satisfied.

In fact in [30], a transformation is presented which automatically synthesizes an efficiently monitorable specification from a *boolean-only* specification. Since the synthesized specification may increase in size, and the monitoring algorithm is linear in *both* the size of the trace and the specification, one must consider whether the transformation is worth it. Further, it is shown that if no partial evaluation for the equations is used, then the converse of presented syntactical characterization also holds. This implies that there are no positive closed walks in an efficiently monitorable specification.

5 Domain-specific LOLA Extensions

In this section, we present the domain-specific LOLA extensions required to improve the applicability in the context of unmanned aircraft. LOLA is a very powerful language and most of the desired properties are expressible as is. However, especially entangled dependencies between several streams increase the size of the specification significantly which, in return, decrease the applicability for usage and performance. Furthermore, LOLA limits the expressiveness of prior knowledge about the domain which specification engineers often have. In the following, we highlight the extended syntactical fragments in order to present the extensions in a concise and understandable way. The complete extended LOLA syntax is listed in Appendix A.

In Syntax 4, we depict the basic structure of our extensions. Possibilities to observe and control the behavior of an online monitor as well as the analysis of offline executions are introduced. Furthermore, we see why prior knowledge of the domain is useful and can improve efficiency. Additionally, to improve the reusability and readability, we propose macro definitions. Macros will facilitate LOLA libraries and, by that, allow the extraction of often used LOLA patterns to an even higher level of abstraction.

$$
\begin{aligned}
\langle \mathit{streamDef} \rangle \quad ::= \; & \langle \mathit{inputDef} \rangle \\
| \; & \langle \mathit{constantDef} \rangle \\
| \; & \langle \mathit{outputDef} \rangle \\
| \; & \langle \mathit{observableBehavior} \rangle \\
| \; & \langle \mathit{knowledge} \rangle \\
| \; & \langle \mathit{macroDef} \rangle
\end{aligned}
$$

Syntax 4: Extended structure

In the following, we formally introduce the extensions. First, the extensions to the stream expressions are presented. Second, we show how we improved the observable monitoring behavior and its controllable behavior. Third, the new statement operator is shown which uses specified prior knowledge. Finally, we propose macros to reduce redundant work while writing specifications.

5.1 Stream Expressiveness

One key feature of LOLA is the computation of statistics. In order to enrich this feature, we extend the supported stream types by *double*, *string*, and *tuple*. *Double* values are required quite naturally due to the monitoring of real time systems. The reason for *string* support is based on the fact that high-level reasoning is often kept

humanly readable to understand the process. High-level reasoning applies to both the system reasoning and the desired specification. The last type which we only allow for output streams is *tuple*.

There are two reasons why *tuples* are interesting. They allow combining streams to improve the structure of specifications. For instance, when computing the average over the trace, the sum is an unavoidable byproduct. The other reason concerns concurrent/distributed LOLA monitoring. Computing the strongly connected components based on the dependency graph allows generating independent LOLA monitors for each of them. In this case, *tuples* could be used to enrich the dependency graph to group streams together which do not explicitly depend on each other according to the dependency graph itself, but should be run on the same LOLA instance.

Syntax 5 shows that among others the notion of functions is introduced. Note that types and functions were never explicitly listed in [22]. In fact, we increased the practical expressiveness and by far not the theoretical. In the consequent subsections, we present the extensions in detail by motivating and defining them formally. For the remainder, we extend the set of types T by *double*, *string*, and *tuple*. Further, the well-typedness and semantics of binary operators are extended for double types in the natural way. The same holds for the string comparison '='. We enforce that both operands of a binary operator are of the same type. We do not offer automatic type conversion. Instead, type conversion is performed via functions. A specification engineer should not lose that control.

$$
\begin{aligned}
\langle expr \rangle \quad ::= \; & \langle literal \rangle \mid \text{`('} \, \langle expr \rangle \, \text{`)'} \mid \langle unaryOp \rangle \\
\mid \; & \langle binaryOp \rangle \mid \langle ifExpr \rangle \mid \langle switchExp \rangle \\
\mid \; & \langle keyword \rangle \mid \langle streamAccess \rangle \mid \langle functionOp \rangle
\end{aligned}
$$

Syntax 5: Expressions

5.1.1 Functions

The allowed functions are given in Syntax 6. Most of their semantics are obvious from their name but to avoid misunderstandings, we formally introduce them next. We extend the notion of *well-typedness* and present their semantics. We abbreviate a tuple entry access similar to trace positions. For instance, $(1, 2, 3)\#1$ would be resolved to 2. We do the same for tuple types.

Definition 8. Well-Typedness Extension: Functions
For *well-typedness*, we extend the rules from Definition 1 in the following way.

- If $e_i \mathrel{\widehat{=}} f(e_j)$ and f is a function with a single parameter then
  - If $f \in \{abs, bin\_to\_int\}$ and e_j is of type *int* then e_i is of type *int*.
  - If $f \in \{abs, sqrt, log, cos, sin, tan, ceil, floor, round\}$ then e_j has to be of type *double* and e_i is of type *double*[3].
  - If $f \in \{int\}$ then e_j has to be of type *double* and e_i is of type *int*.
  - If $f \in \{double\}$ then e_j has to be of type *int* and e_i is of type *double*.
  - If $f \in \{length\}$ then e_j has to be of type *string* and e_i is of type *int*.
- If $e_i \mathrel{\widehat{=}} f(e_j, e_k)$ and f is a function with two parameters then
  - If $f \in \{atan2\}$ then both e_j and e_k have to be of type *double* and e_i is of type *double*.
  - If $f \in \{difference\}$ then both e_j and e_k have to have the same type, either *double* or *int* and respectively the type of e_i is *double* or *int*.
  - If $f \in \{contains, startswith, endswith\}$ then e_j and e_k have to be of type *string* and e_i is of type *bool*.
  - If $f \in \{get\}$ then e_j has to be of type *tuple*, e_k of type *int*, and e_i has the same type as $e_j \# e_k$
  - If $f \in \{combine\}$ then e_j and e_k have to be of type *tuple* and e_i has type (e_j, e_k).
- If $e_i \mathrel{\widehat{=}} f(e_1, \ldots, e_n)$ and f is a function with $n \in \mathbb{N}$ many parameters then
  - If $f \in \{max, min\}$ then all parameters have to be of the same type *int* or *double* and e_i is of type *int* or *double* respectively.
  - If $f \in \{equals\}$ then all parameters have to be of type *string* and e_i is of type *bool*.
  - If $f \in \{concat\}$ then all parameters have to be type *string* and e_i is of type *string*.
  - If $f \in \{extract\}$ then e_1 has to be of type *tuple* and all other parameters have to be of type *int* and e_i is of type $(e_1 \# e_2, \ldots e_1 \# e_n)$.

Definition 9 extends the LOLA semantics by functions. Since functions do not change the dependency graph, there is no need for updating its definition.

[3]Note, $sqrt(a, b)$ and $log(a, b)$ abbreviate $a \hat{} (1/b)$ respectively $log(a)/log(b)$

Definition 9. Lola Semantic Extension: Functions
We extend the previously introduced function *eval* which, given an expression and a position in the trace returns the result of the expression. Again, let e_l, e_r, and e_c be well-typed subexpressions.

- Number functions:
  - $eval(abs(e_l))(j) = |eval(e_l)(j)|$
  - $eval(atan2(e_l, e_r))(j) = atan2(eval(e_l)(j), eval(e_r)(j))$
  - $eval(difference(e_l, e_r))(j) = |eval(e_l)(j) - eval(e_r)(j)|$
  - $eval(max(e_1, \ldots, e_n))(j) = max\{eval(e_1)(j), \ldots, eval(e_n)(j)\}$, where $n \in \mathbb{N}$ and $e_1, \ldots, e_n$ are valid subexpressions.
  - $eval(min(e_1, \ldots, e_n))(j) = min\{eval(e_1)(j), \ldots, eval(e_n)(j)\}$, where $n \in \mathbb{N}$ and $e_1, \ldots, e_n$ are valid subexpressions.
  - $eval(sqrt(e_l))(j) = \sqrt{eval(e_l)(j)}$
  - $eval(log(e_l))(j) = \log(eval(e_l)(j))$
  - $eval(cos(e_l))(j) = \cos(eval(e_l)(j))$
  - $eval(sin(e_l))(j) = \sin(eval(e_l)(j))$
  - $eval(tan(e_l))(j) = \tan(eval(e_l)(j))$
  - $eval(bin\_to\_int(e_l))(j) = \begin{cases} int(eval(e_l)(j)) & \text{, if eval(e\_l)(j) is a valid binary} \\ error & \text{, otherwise.} \end{cases}$
  - $eval(int(e_l))(j) = int(eval(e_l)(j))$
  - $eval(double(e_l))(j) = double(eval(e_l)(j))$
  - $eval(ceil(e_l))(j) = \lceil eval(e_l)(j) \rceil$
  - $eval(floor(e_l))(j) = \lfloor eval(e_l)(j) \rfloor$
  - $eval(round(e_l))(j) = \lfloor (eval(e_l)(j)) + 0.5 \rfloor$
- String functions:
  - $eval(contains(e_l, e_r))(j) = eval(e_l)(j)$ is contained in $eval(e_r)(j)$
  - $eval(equals(e_1, \ldots, e_n))(j) = \begin{cases} true & \text{, if forall 1<i<n, eval(e\_i)(j)=eval(e\_i+1)(j)} \\ false & \text{, otherwise.} \end{cases}$
    , where $n \in \mathbb{N}$ and $e_1, \ldots, e_n$ are valid subexpressions.
  - $eval(startswith(e_l, e_r))(j) = eval(e_l)(j)$ starts with $eval(e_r)(j)$

  - $eval(endswith(e_l, e_r))(j) = eval(e_l)(j)$ ends with $eval(e_r)(j)$
  - $eval(concat(e_l, e_r))(j) = eval(e_l)(j)$ concatenated with $eval(e_r)(j)$
  - $eval(length(e_l))(j) =$ length of $eval(e_l)(j)$
- Tuple functions:
  - $eval(get(e_l, e_r))(j) = eval(e_l)(j) \# eval(e_r)(j)$
  - $eval(extract(e_1, \ldots, e_n))(j) =$
    $(eval(e_1)(j) \# eval(e_2)(j), \ldots, eval(e_1)(j) \# eval(e_n)(j)$
    , where $n \in \mathbb{N}$ and $e_1, \ldots, e_n$ are valid subexpressions.
  - $eval(combine(e_l, e_r))(j) = (eval(e_l)(j), eval(e_r)(j))$

⟨*functionOp*⟩ ::= ⟨*numberFunction*⟩ | ⟨*stringFunction*⟩ | ⟨*tupleOp*⟩

⟨*numberFunction*⟩ ::= `'abs'` `'('` ⟨*expr*⟩ `')'` | `'difference'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`
| `'max'` `'('` ⟨*expressions*⟩ `')'` | `'min'` `'('` ⟨*expressions*⟩ `')'`
| `'sqrt'` `'('` ⟨*expr*⟩ **[**`','` ⟨*doubleLiteral*⟩**]**`')'`
| `'log'` `'('` ⟨*expr*⟩ **[**`','` ⟨*doubleLiteral*⟩**]** `')'`
| `'cos'` `'('` ⟨*expr*⟩ `')'` | `'sin'` `'('` ⟨*expr*⟩ `')'`
| `'tan'` `'('` ⟨*expr*⟩ `')'` | `'bin_to_int'` `'('` ⟨*expr*⟩ `')'`
| `'int'` `'('` ⟨*expr*⟩ `')'` | `'double'` `'('` ⟨*expr*⟩ `')'`
| `'ceil'` `'('` ⟨*expr*⟩ `')'` | `'floor'` `'('` ⟨*expr*⟩ `')'`
| `'round'` `'('` ⟨*expr*⟩ `')'`

⟨*stringFunction*⟩ ::= `'contains'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`
| `'equals'` `'('` ⟨*expressions*⟩ `')'`
| `'startswith'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`
| `'endswith'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`
| `'concat'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`
| `'length'` `'('` ⟨*expr*⟩ `')'`

⟨*tupleOp*⟩ ::= `'get'` `'('` ⟨*expr*⟩ `','` ⟨*intLiteral*⟩ `')'`
| `'extract'` `'('` ⟨*expr*⟩ `','` ⟨*intLiteral*⟩ **[**`','` ⟨*intLiteral*⟩**]*** `')'`
| `'combine'` `'('` ⟨*expr*⟩ `','` ⟨*expr*⟩ `')'`

Syntax 6: Functions

The following example illustrates how we can use functions to improve the conciseness of our specification.

Example 6. Maximal Velocity
Consider Listing 7, let *vel_vert* and *vel_hori* be input streams indicating the vertical, respectively horizontal, velocity of an UAV. A negative vertical velocity indicates that the UAV is descending and a negative horizontal velocity represents that the UAV is flying backwards. An interesting property of the system, both offline and online, might be the maximal velocity independent of the direction. Using the *max* function on the *abs*olute values of the inputs, it is straight forward to express this property. Note that we require the previous *m_vel* to carry the maximal value along the trace. Otherwise, the output stream would only represent the local maximum in each position in the trace. Further, without using the *max* function, we had to use several if-statements, increasing the size of the specification and their error susceptibility. In Section 5.4, we propose a generalization of this idea.

```
1 input  double vel_vert, velo_hori
2 output double m_vel := max( abs(vel_vert), abs(velo_hori), m_vel[-1, 0.0])
```

Listing 7: A Lola specification that calculates the maximal velocity.

5.1.2 Keywords

The next extension to Lola we introduce are keywords. We currently allow `position` (current position in the trace), `last_position` (length of the trace), `int_max` (maximal integer value), `int_min` (minimal integer value), `double_max` (maximal double value), `double_min` (minimal double value). `position` is often used for statistical computations, e.g. average, and `last_position` is interesting for offline monitoring since the knowledge about the last position is easy to obtain and useful for specifications, e.g. in the middle of the trace some property should hold. Note that in Listing 8, a trigger might be raised in the middle of the trace during offline monitoring, whereas in online monitoring, the earliest notification will be given after termination. Of course, other keywords are possible, but we restricted ourselves to this specific set since others did not increase the expressiveness. Instead, much more the possibilities to express the same property. We formally extend well-typedness and the semantics of keywords in the following.

```
1 input  int    values
2 output int    sum          :=  values + sum[-1, 0]
3 output double average      :=  double(sum) / (double(position+1))
4 output bool   mid_exceed   := (position=(last_position / 2)) & (average>100.0)
5 trigger mid_exceed
```

Listing 8: A Lola specification that checks that average in the middle of trace is greater than 100.

Definition 10. Well-Typedness Extension: Keywords
For *well-typedness*, we extend the rules from Definition 1 in the following way.

- If $e_i \mathrel{\widehat{=}} position$ then e_i is of type *int*.
- If $e_i \mathrel{\widehat{=}} last\_position$ then e_i is of type *int*.
- If $e_i \mathrel{\widehat{=}} int\_max$ then e_i is of type *int*.
- If $e_i \mathrel{\widehat{=}} int\_min$ then e_i is of type *int*.
- If $e_i \mathrel{\widehat{=}} double\_max$ then e_i is of type *double*.
- If $e_i \mathrel{\widehat{=}} double\_min$ then e_i is of type *double*.

Definition 11. Lola Semantic Extension: Keywords
We extend the previously introduced function *eval* which, given an expression and a position j in the trace returns the result of the expression. Let N be the length of the trace.

- $eval(position)(j) = j$
- $eval(last\_position)(j) = N - 1$
- $eval(int\_max)(j) =$ largest integer value
- $eval(int\_min)(j) =$ smallest integer value
- $eval(double\_max)(j) =$ largest double value
- $eval(double\_min)(j) =$ smallest double value

5.1.3 Absolute Access

So far, we only allow relative stream accesses. The *relative offset operator* accesses the previous/future value of a stream depending on the current position in the trace. In this subsection, we introduce and motivate the use of the *absolute offset operator*.

Imagine, we only have access to the current absolute height *hgt* (above sea level) via an input stream. The allowed flight height of UAV missions can be relatively restricted, e.g. do not raise or descent by more than 5 meters. Such relative restrictions require capturing the initial state of the UAV to refer to it later in the flight analysis. Furthermore, since we could start a flight in the valley and the other time in the mountains it is not possible to assume that the initial state remains the same.

There are two obvious ways of capturing this initial state: we could define constants to specify the current state. This would yield efficient monitoring but would raise the problem, that whenever we change the location we had to adjust each of these possibly many constants. Further, there would never be a specification which is proven/assumed correct since every change in the specification might introduce new errors. The other approach is storing the initial state using streams. Listing 9 shows this approach. Using the keyword `position`, we can initialize the *init_hgt* stream with the initial height. Afterwards, this height can be propagated over the entire trace via the offset operator. Notice that this approach involves re-evaluating a stream which actually represents a constant stream apart from the initialization of the first position.

```
1 const  double bound    := 5.0
2 input  double hgt
3 output double init_hgt := if position = 0 { hgt } else { init_hgt[-1, 0.0] }
4 output bool   exceeds  := difference(init_hgt, hgt) > bound
5 trigger exceeds
```

Listing 9: A Lola specification that calculates the current relative height using an auxiliary stream.

In order to interpolate between constant and dynamic values, we introduce the *absolute offset operator*. Its syntax is given in Syntax 7.

Similar to the relative offset operator, the well-typedness follows from the fact that the type of stream itself and the type of its out-of-bounds value have to match.

⟨*streamAccess*⟩ ::= ⟨*identifier*⟩ **[** ⟨*relativePos*⟩ | ⟨*absolutePos*⟩ **]**

⟨*absolutePos*⟩ ::= '`#`' ⟨*offset*⟩ | '`#`' ⟨*window*⟩

⟨*offset*⟩ ::= '`[`' ⟨*offsetValues*⟩ '`,`' ⟨*oobV*⟩ '`]`'

⟨*window*⟩ ::= '`[`'⟨*offsetValues*⟩'`..`'⟨*offsetValues*⟩'`,`'⟨*oobV*⟩'`,`'⟨*binaryOp*⟩'`]`'

⟨*offsetValues*⟩ ::= **(** ⟨*intLiteral*⟩ | '`last_position`' **)**

Syntax 7: Absolute Offset Operator

Definition 12. Well-Typedness Extension: Absolute Offset Operator
For *well-typedness*, we extend the rules from Definition 1 in the following way.

- If $e_i \mathrel{\widehat{=}} e_j \# [n,\ l]$ then e_i and e_j have to be of the same type t_j and, further, l has to be either a *literal* or a *constant* of type t_j. Additionally, n is either a positive integer or a *keyword* representing an integer greater or equal zero.

The semantics of the absolute offset operator is stated in the following definition.

Definition 13. LOLA Semantic Extension: Absolute Offset Operator
We extend the previously introduced function *eval* which, given an expression and a position in the trace returns the result of the expression as follows:

$$eval(e\#[p, oobv])(j) = \begin{cases} eval(e)(p) & \text{, if } 0 \leq eval(p)(j) < N \\ eval(oobv)(j) & \text{, otherwise} \end{cases}$$

, where N is the length of the trace, j is any position, $p \in \mathbb{N}$, and *oobv* represents the out-of-bounds value.

Notice that specifying values at the "end" of the trace is highly inefficient for online monitoring. Observe further that Definition 3 sufficiently captures the temporal dependency of absolute offsets, i.e. we do not require any extensions to the graph to check well-formedness.

Listing 10 shows that we are now able to specify the previous property concisely and efficiently. There are neither extra constants nor extra streams involved which store the initial state values.

```
1 const  double bound    := 5.0
2 input  double hgt
3 output bool   exceeds  := difference(hgt#[0, 0.0], hgt) > bound
4 trigger exceeds
```

Listing 10: A Lola specification calculates the current relative height without an auxiliary stream.

In the context of offset operators, we further introduce windowing for both the relative and the absolute offset operator. The windowing: `a [i..j, oobv, ○]` is an abbreviation for `a[i, oobv]` $\circ \cdots \circ$ `a[j, oobv]` where `i,j` $\in \mathbb{N}$, `i` $<$ `j`, `a` is a stream and $\circ$ is an available binary operator. Binary computations are unfolded using the operator precedence whereas binary comparisons are performed pairwise.

5.1.4 Frozen Stream Values

In the previous subsection, we have introduced the notion of windowing for both relative and absolute offsets. This is very useful when checking whether a value along a trace is frozen, i.e. does not change. Consider Listing 11 and assume an arbitrary sensor for which it is very unlikely that the value remains the same. In the listing, we use normal windowing with a window size of 3. Unfortunately, this check is highly inefficient since the windows overlap and intermediate values are computed all over again although they do not change. For instance in the first position, the values *sensor#1, sensor#2,* and *sensor#3* are compared in the next step we compare *sensor#2, sensor#3,* and *sensor#4*.

```
1 const  int    w_size     := 3
2 input  double sensor
3 output bool   is_frozen  := sensor[1..w_size, false, =]
4 trigger is_frozen
```

Listing 11: A Lola specification that checks whether the sensor is frozen, using a normal window.

To overcome this limitation, we introduce *frozen offsets* and *frozen windows* for the relative offset operator[4]. Syntax 8 depicts the extensions.

[4]Absolute offsets are frozen by design.

⟨*streamAccess*⟩ ::= ⟨*identifier*⟩ **[** ⟨*relativePos*⟩ | ⟨*absolutePos*⟩ **]**

⟨*relativePos*⟩ ::= ⟨*offset*⟩ | ⟨*frozenOffset*⟩
| ⟨*window*⟩ | ⟨*frozenWindow*⟩

⟨*frozenOffset*⟩ ::= '`[`' ⟨*offsetValues*⟩ '`,`' ⟨*oobV*⟩ '`,`' ⟨*frozenTime*⟩ '`]`'

⟨*frozenWindow*⟩ ::= '`[`' ⟨*offsetValues*⟩ '`..`' ⟨*offsetValues*⟩ '`,`'
⟨*oobV*⟩ '`,`' ⟨*binaryOp*⟩ '`,`' ⟨*frozenTime*⟩ '`]`'

⟨*frozenTime*⟩ ::= ⟨*identifier*⟩ | ⟨*literal*⟩

Syntax 8: Frozen Offsets and Frozen Windows

The intuition is to express the property of *is_frozen* by a window which takes single large discrete shifts instead of several small shifts, in each subsequent position. The additional parameter and last parameter in brackets indicates how long a value is frozen. With the new operators, we can write the formula as:
`output bool is_frozen := sensor[1..w_size, false, =, w_size-1]`
which compares the first three positions, i.e. we evaluate whether *sensor#1, sensor#2,* and *sensor#3* are the same only at the fourth position in the trace, we shift the window to check the equality for *sensor#3, sensor#4, sensor#5* and so on. An algorithm can make use of this structure, which improves the performance of the monitor. Note that the *frozen offsets* and *frozen windows* differ from the standard relative offset operator only by the extra last value representing the *frozenTime*.

Definition 14. Well-Typedness Extension: Frozen Offset and Frozen Window
For *well-typedness*, we extend the rules in Definition 1 as follows:

- If $e_i \mathrel{\widehat{=}} e_j\ [n,\ l,\ ft]$ then e_i and e_j have to be of the same type t_j and, further, l has to be either a *literal* or a *constant* of type t_j. Additionally, each n and ft are either a positive integer or a *keyword*, or respectively a *constant* representing an integer.

A frozen window `a [i..j, oobv, ∘, ft]` is an abbreviation for:
`a[i, oobv, ft]` $\circ \cdots \circ$ `a[j, oobv, ft]` where `i,j` $\in \mathbb{N}$, `i` $<$ `j`, `a` is a stream and $\circ$ is an available binary operator. Binary computations are unfolded using the operator precedence whereas binary comparisons are performed pairwise.

Definition 15. Lola Semantic Extension: Frozen Offset and Frozen Window
We extend the previously introduced function *eval* which, given an expression and a position in the trace returns the result of the expression. Let p be a keyword or a literal representing an integer, *oobv* a constant or a literal representing the out-of-bounds value, and ft a constant or a literal representing an integer interpreted as the frozen time. Further, let j be the current position in the trace with total length N and let *frozenDuration* be a function which takes j and *ft* and computes the duration a value is still frozen, i.e. $j - (j \ \% \ (ft + 1))$, then:

- if $frozenDuration(j, ft) = 0$:
  $eval(e[p, oobv, ft])(j) = \begin{cases} eval(e)(j + eval(p)(j)) & \text{, if } 0 \leq eval(p)(j) < N \\ eval(oobv)(j) & \text{, otherwise} \end{cases}$
- otherwise:
  $eval(e[p, oobv, ft])(j) = eval(e[p, oobv, ft])(j - 1)$

Observe that the given definition does not distinguish between out-of-bounds values and actual values when it comes to deciding whether to take the frozen value or the current new value. Further, Definition 3 suffices again to capture the temporal dependency, i.e. well-formedness can still be checked using the dependency graph.

5.2 Prior Knowledge

Domain experts have some kind of prior knowledge of the domain or the process itself. For instance, the expert knows that the fuel tank only decreases during a mission. Process knowledge might involve insight of the event sequence used by a mission manager, e.g. a mission starts with a TAKEOFF event and ends with a LAND event. Syntax 9 shows the expressible knowledge. Using the dependency graph, it is possible to automatically derive some of the expressible knowledge. Below, we present the ones which can be derived automatically and indicate how to do so:

- `past_only`: A given stream s can be evaluated independent of the future.
  - There is no walk starting from s containing a positive weighted edge.
- `future_only`: A given stream s can be evaluated independent of the past.
  - There is no walk starting from s containing a negative weighted edge.
- `efficient_fragment`: A given specification is efficiently monitorable.
  - Using Definition 6.
- `inefficient_fragment`: A given specification is not efficiently monitorable.
  - Using Definition 6.

- `evaluation_order`: The order in which the outputs are evaluated.
  - Use the dependency graph to schedule the outputs.

At first glance, the dual operators might seem unnecessary, e.g. `future_only`, but we want to remind here that offline monitoring allows evaluating the trace backwards where this is useful. The remaining knowledge operators are `monotone_inc` and `monotone_dec`. The operator `monotone_inc` and `monotone_dec` state that the values of a stream are monotonically increasing or decreasing, respectively. It is not possible to derive this knowledge automatically since the behavior of an input stream is unknown to a LOLA monitor.

⟨*priorKnowledge*⟩ ::= ⟨*streamKnowledge*⟩ | ⟨*specificationKnowledge*⟩

⟨*streamKnowledge*⟩ ::= '`monotone_inc`' ⟨*identifier_list*⟩
| '`monotone_dec`' ⟨*identifier_list*⟩
| '`past_only`' ⟨*identifier_list*⟩
| '`future_only`' ⟨*identifier_list*⟩

⟨*specificationKnowledge*⟩ ::= '`efficient_fragment`'
| '`inefficient_fragment`'
| '`evaluation_order`' ⟨*identifier_list*⟩

Syntax 9: Prior Knowledge

We argue that this knowledge can be used by operators to improve the performance, especially in the case of future dependencies. Syntax 10 gives the structure of such a new *switch* statement operator which makes use of prior stream knowledge. The keyword *position* is by default set to monotonically increasing.

Example 7. Flight Modes
Consider Listing 12. The flight commands during a mission are provided via the integer stream *enum_cmd* and the stream *remote_control* indicates whether the system changes to remote (assumed once). In the example, we abstract from the explicit properties checked in the case blocks. However, we assume that depending on the current state of the system the property can be split into three: initialization phase(0), flight phase(1), and landing phase(2). This grouping is done via the *monotonically increasing* output *cnd* which is used as the switch condition in *check*. The stream *cnd* has an additional group(3) which distinguishes whether the system changes to remote control within the next three steps.

Informally, the switch statement encodes an acyclic state machine whenever the condition is flagged as *monotone_inc* or *monotone_dec*. The states of the machine are given by the case blocks and the initial state is assumed to be the first case block. The switch operator makes use of the fact that whenever it descends to a higher case block (the default block is always the highest) it knows by *monotone_increasing* that it will never check lower case blocks again[5]. This is not only useful when we splitting up the property into many cases but also when the switch condition depends on the future like in the example. In such a case, we only need to update the interesting cases and do not keep track of the others.

```
1  input  int   enum_cmd
2  input  bool  remote_control
3  output int   cnd := if remote_control[0..3,false,|,3] | cnd[-1,0] = 3
4    {3}  else {  if enum_cmd <= 0 {0} elif enum_cmd >= 10 {2} else {1}  }
5  monotone_inc cnd
6  output bool  check :=  switch cnd {
7                          case 0 {  ...*initialization* ...  }
8                          case 1 {  ...*flight commands*...  }
9                          case 2 {  ...*landing*        ...  }
10                         default{ false }
11                         }
12 trigger check
```

Listing 12: An abstracted Lola specification with a switch which uses a monotone increasing stream as condition.

In the remainder of this subsection, we formally define the switch operator.

Definition 16. Well-Typedness Extension: Switch Operator
Let $l \in \mathbb{N}$, e_{cnd}, e_{c_1}, ..., e_{c_l}, and e_d be well-typed subexpressions and c_1, ..., c_l be distinct literals. For *well-typedness*, we extend the rules from Definition 1 in the following way.

- If $e_i \mathrel{\widehat{=}}$ switch $e_{cnd}\{\text{case } c_1\{e_{c_1}\} \ldots \text{case } c_l\{e_{c_l}\} \text{ default } \{e_d\}\}$ then e_{cnd}, c_1, ..., and c_l have to be of the same type and e_{c_1}, ..., e_{c_l}, and e_d have to be of the same type, finally e_i is of type e_d.
  Additionally, if e_{cnd} represents a stream with or without an offset then whenever this stream is flagged as monotonically increasing: c_1, ..., c_l have to be ordered increasingly respectively decreasingly for monotonically decreasing.

[5]Note that we enforce that the case blocks are ordered.

The order of integers and doubles is given by $<$, strings are ordered lexicographically, *true* is greater than *false*, and tuples are ordered by pairwise recursively comparing their entries starting with position zero.

Definition 17. Lola Semantic Extension: Switch Operator
We extend the previously introduced function *eval* which given an expression and a position in the trace returns the result of the expression. Let *state* be a variable. This variable captures the case block to which the latest condition (relative to the trace) evaluated to, initially $undefined$. If the switch condition refers to a stream which is monotonically increasing or decreasing then *state* is set to the first case block if it exists otherwise to the default block.

- if *state* is $undefined$:
  $eval($ switch $e_{cnd}\{$ case $c_1\{e_1\} \ldots$ case $c_l\{e_{c_l}\}$ default $\{e_d\}\})(j)$
  $$= \begin{cases} eval(e_1)(j) & \text{, if } eval(e_{cnd})(j) = c_1 \\ \ldots & \\ eval(e_{c_l})(j) & \text{, if } eval(e_{cnd})(j) = c_l \\ eval(e_d)(j) & \text{, otherwise} \end{cases}$$
- else:
  1. $eval($switch $e_{cnd}\{$case $c_{state}\{e_{state}\} \ldots$ case $c_l\{e_{c_l}\}$ default $\{e_d\}\})(j)$
     $$= \begin{cases} eval(e_{state})(j) & \text{, if } eval(e_{cnd})(j) = c_{state} \\ \ldots & \\ eval(e_{c_l})(j) & \text{, if } eval(e_{cnd})(j) = c_l \\ eval(e_d)(j) & \text{, otherwise} \end{cases}$$
  2. If condition is *monotonically increasing* and c_{state} is smaller than $eval(e_{cnd})(j)$ according to the case order then change *state* to the current case/default block, respectively c_{state} is greater than $eval(e_{cnd})(j)$ for a *monotonically decreasing* condition.

Well-typedness assures that whenever we set *state* to a case/default block, we never check previous cases again. Additionally, considering well-formedness, since the switch operator does not change the temporal dependencies, we do not have to adjust anything.

⟨*switchExpr*⟩ ::= '`switch`' ⟨*expr*⟩ '`{`' ⟨*cases*⟩ ⟨*default*⟩ '`}`'

⟨*cases*⟩ ::= ϵ
| '`case`' ⟨*literal*⟩ '`{`' ⟨*expr*⟩ '`}`'
| '`case`' ⟨*literal*⟩ '`{`' ⟨*expr*⟩ '`}`' ⟨*cases*⟩

⟨*default*⟩ ::= '`default`' '`{`' ⟨*expr*⟩ '`}`'

Syntax 10: Switch Statement

5.3 Observable Monitoring Behavior

In this section, we improve the interface between the user and Lola. We extend the notifications a user receives, allow to generate new offline logs, and grant control over the Lola evaluation. Syntax 11 contains the syntactical extensions. The overall order of the observable monitor behaviors evaluation is: 1.) Online Behavior, 2.) Offline Behavior, 3.) Control Behavior.

5.3.1 Online Feedback

Whenever Lola is running (online), *trigger*s are the only possibility to give feedback on the current trace evaluation, i.e. whether an error occurred or an interesting threshold is exceeded. Previously, we did not mention the details of triggers and, in fact, there exist different interpretations. In [22], a trigger is a special kind of a boolean stream, whereas the old implementation presented in Section 4, interprets them as a flag on a boolean stream. We decided to use the latter. We allow multiple flags on a stream to facilitate several different triggers on a single stream. Furthermore, it is possible to declare a trigger on an input stream without an additional stream simulating its value. Since we are using flags further modifications to the stream are not expressible anymore. The syntax of a trigger consists of three parts: a kind, a well-typed condition, and an optional message. The condition is either a valid stream $s \in S$ or a comparison between a stream $s \in S$ and a literal $l \in L$. We call the condition well-typed whenever the kind is `print`, s is a boolean stream, or the comparison is well-typed. We give its formal semantics in Definition 18.

⟨*observableBehavior*⟩ ::= ⟨*onlineBehavior*⟩ **[**'`with`' ⟨*stringLiteral*⟩ **]**
| ⟨*offlineBehavior*⟩ '`at`' ⟨*location*⟩
| ⟨*controlCommands*⟩ **[** '`with`' ⟨*stringLiteral*⟩ **]**

⟨*onlineBehavior*⟩ ::= '`trigger`' ⟨*condition*⟩
| '`trigger_once`' ⟨*condition*⟩
| '`trigger_change`' ⟨*condition*⟩
| '`print`' ⟨*identifier*⟩
| '`snapshot`' ⟨*condition*⟩

⟨*offlineBehavior*⟩ ::= '`filter`' ⟨*identifier_list*⟩ '`if`' ⟨*condition*⟩
| '`tag`' '`as`' ⟨*identifier_list*⟩ '`if`' ⟨*condition*⟩
'`with`' ⟨*identifier_list*⟩

⟨*controlCommands*⟩ ::= '`exit`' ⟨*condition*⟩
| '`pause`' ⟨*condition*⟩
| '`reset`' ⟨*condition*⟩

⟨*condition*⟩ ::= ⟨*identifier*⟩ **[** ⟨*comparison*⟩ ⟨*literal*⟩ **]**

Syntax 11: Observable Behavior

Definition 18. LOLA Semantic Extension: Online Behavior
Given the evaluation function *eval*, the current position in the trace j, and a declared observable online behavior:
`obs_kind condition message`, where `obs_kind` $\in$ { `trigger, trigger_once, trigger_change, print, snapshot`} and `message` is an arbitrary string. Additionally, let $s \in S$ be the stream referred in the `condition`, *eval_cnd* be the current condition evaluation, i.e. *eval*(`condition`)(j), and let *last_obs* be the value of the `condition` evaluation which results in the last notification (i.e. prior to j), initially set to *undefined*. Furthermore, let *streamvalue* be a function which takes a stream and prints its current value, i.e. undefined if it cannot be resolved, yet. Then, we define its semantics as follows:

- `obs_kind` is `trigger`:
  if *eval_cnd* holds
  then notify user by: *streamvalue(s)* and print(`message`).

- `obs_kind` is `trigger_once`:
  if *eval_cnd* holds and *last_obs* is undefined
  then notify user by: *streamvalue(s)* and print(`message`).
  and set *last_obs* to *eval_cnd*.

- `obs_kind` is `trigger_change`:
  if either *last_obs* is undefined and *eval_cnd* holds
  or *last_obs* is defined and *eval_cnd* unequal to *last_obs*
  then notify user by: *streamvalue(s)* and print(`message`).
  and set *last_obs* to *eval_cnd*.

- `obs_kind` is `print`:
  if *true*
  then notify user by: *streamvalue(s)* and print(`message`).

- `obs_kind` is `snapshot`:
  if *eval_cnd* holds
  then notify user by: $\forall s' \in S$, *streamvalue(s')* and print(`message`).

`trigger, trigger_once, trigger_change, print` can be used to give the user feedback about a single stream. `snapshot` offers the user online feedback about the current state of LOLA, i.e. current statistical measurements.

Example 8. Observable online monitor behaviors
Figure 7 illustrates the different granularities of user feedback available. The input stream *check* is used as *condition* for the listed online behaviors below. The notification *print* reports the complete stream whereas a *trigger* reports only the

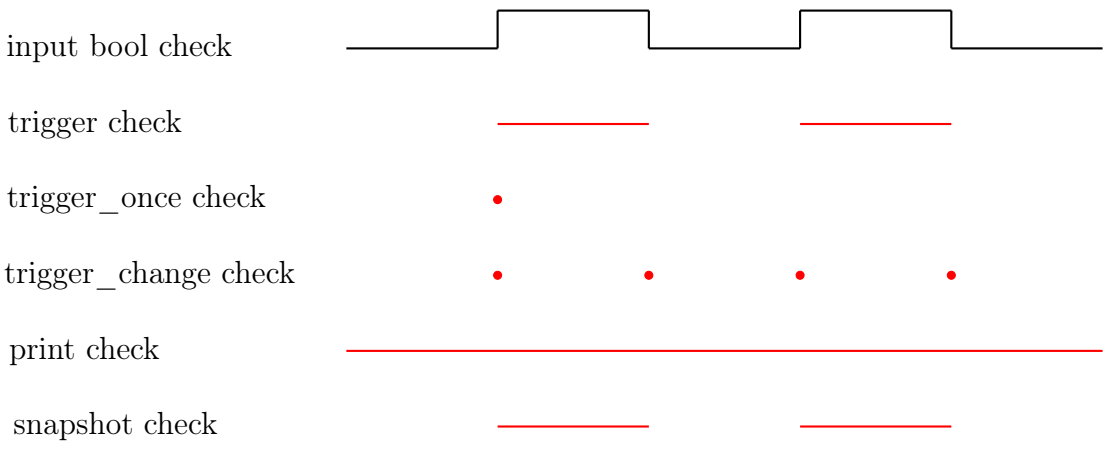


Figure 7: We specify all kinds of online feedback on the given boolean input stream check. For each online feedback, we depict its behavior whenever its condition changes.

positions in the trace where the stream evaluates to true. In case we raise too many triggers, *trigger_change* can be used to notify the entrance into a trace sequence where the condition holds at its exit. *trigger_once* signals once the condition holds since this might already suffice. Note that there is a difference in the feedback between a *trigger* and a *snapshot*. The *snapshot* is quantified over all the existing output streams whereas a *trigger* is dedicated to a single stream.

5.3.2 Offline Feedback

In offline analysis, where engineers want to deepen their understanding on how the sensor inputs/outputs evolve over time and which properties hold, statistical measurements and error detection are not the only features LOLA offers. Currently, engineers use plots for identifying errors by manual inspection and for understanding how an occurring error can be fixed. Arguing over several such plots to identify an error is hard, especially for long traces where the error occurs only in a combination of several plots in different local timestamps. The properties LOLA streams offer might not suffice to give enough information about the problem source but, still, LOLA is able to identify the fault. We augmented LOLA by *tagging* which can be used to produce a new data file based on stream values. For instance, this new file can be a filtered version of the original data and can later be used by engineers to plot this filtered data only. The Syntax 11 (above) of *tagging* involves a well-typed boolean filter condition and lists of identifiers which represent *past_only* streams. We give its semantics in Definition 19.

Definition 19. Lola Semantic Extension: Offline Behavior
Given the evaluation function *eval*, the current position in the trace j, and a declared observable offline monitor behavior *obs*, let *cnd* be the condition used to filter the data and S be the set of all streams. Furthermore, let $s_1 \in S, ..., s_n \in S$ and *write* be a function which takes a list of evaluated streams, a list of the corresponding names, and writes their current values to a location l. $s_{new_1}, ..., s_{new_n}$ are arbitrary but pairwise distinct new stream-names. Then, we define the semantics of `tag as` $s_{new_1}, ..., s_{new_n}$ `if` *cnd* `with` $s_1, ..., s_n$ `at` l as follows
if $eval(cnd)(j)$ then $write(eval(s_1)(j), s_{new_1}, l) \ldots write(eval(s_n)(j), s_{new_n}, l)$
The other kind of offline behavior we allow is *filtering* which is a special case of *tagging*. Is is defined as follows:
`filter` $s_1, ..., s_n$ `if` *cnd* `at` l $:=$
`tag as` $s_1, ..., s_n$ `if` *cnd* `with` $s_1, ..., s_n$ `at` l

Example 9. Filtering of data exceeding a given bound
Consider Listing 13. We receive the current state, velocity, and the commanded acceleration. Assume the system can be in a state where it has to *wait* for further commands. This state requires that there is no commanded acceleration and the velocity is close to zero. In post flight analysis, each input stream is individually plotted and an engineer has to manually inspect these plots to find violations. This task is difficult, since a violation is based on values of different plots, which might be temporally shifted. Therefore, especially, entangled errors are hard to find and are often overlooked. Lola can be used in such cases to directly identify the violation if its properties are known, otherwise *tagging* facilitates to narrow down the failure point by filtering the trace. In this example, we do not only filter the data but also enriched it by the average acceleration and renamed the new columns.

```
1 input  string state
2 input  double vel, acc
3 output double avg_acc := switch position < 2
4           { case false { 0.0 }   default { acc[-2..0,0.0,+,3] / 3.0 }
5 output bool   tag_cnd := state = "WaitCmd" & avg_acc > 0.0 & abs(vel) < 1.0
6 tag as   cur_state, cur_vel, cur_acc, avg_acc     if tag_cnd
7   with       state,     vel,     acc, avg_acc     at "Desktop"
```

Listing 13: A Lola specification that uses tagging for invalid command executions.

5.3.3 Control Commands

Simulating flights and testing in general is a very time intensive task. Simulation can run for hours and is only aborted by unsatisfied assertions in the code. Extending Lola by *commands* allows to *define* more sophisticated properties based on which a test or a simulation can be aborted. As before, *commands* are interpreted as flags on streams this time each stream can only be flagged by a single command. A command requires a boolean condition, when to execute the command, and allows an optional message. We support three different types of controls which are executed whenever their condition holds:

- `exit`: Terminates the Lola application.
- `pause`: Pauses the Lola application.
- `reset`: Resets the Lola streams to their initial state.

The order in which we execute the commands is: 1.) `exit`, 2.) `pause`, 3.) `reset`. We chose this order because we think that `exit` might be urgent in case a monitor hinders the system to keep up its pace which might harm the safety guarantees of the system. The reason to arrange `pause` before `reset` is based on the fact that `pause` is reversible compared to `reset`. Controls allow Lola to be controlled via input streams. As an example, consider Listing 14.

```
1 input bool stop_lola
2 output int current_pos := position
3 exit stop_lola
```

Listing 14: A Lola specification that aborts its execution in case of a stop signal.

5.4 Macro Proposal

In this subsection, we motivate why we propose macros as an additional extension. The proposed syntax is given in Syntax 12. There are two main reason why we think macros are useful. First, they allow to encapsulate common used Lola patterns in specifications and, second, bloated constructs can be further abstracted to reuse them in an simpler way.

Introducing new Lola functions
Lola supports a lot of basic functions. In Listing 15, we demonstrate the computation of the distance between two points in a three dimensional space using macros.

⟨*macroDef*⟩ ::= ‘`macro`’ ⟨*identifier*⟩ ‘`(`’ ⟨*typed_parameterList*⟩‘`)`’
‘`{`’ ⟨*macroBody*⟩ ‘`}`’

⟨*typed_parameterList*⟩ ::= ϵ
| ‘`stream`’ ⟨*type*⟩ ⟨*identifier*⟩
| ‘`expression`’ ⟨*expr*⟩
| ⟨*type*⟩ ⟨*identifier*⟩
| ⟨*type*⟩ ⟨*identifier*⟩ ‘`,`’ ⟨*typed_parameterList*⟩

⟨*macroBody*⟩ ::= ϵ | ⟨*macroBody*⟩ ⟨*lola-format*⟩

Syntax 12: Macros

Since each domain often requires some common computations, macros are a simple way to introduce new functions to Lola. The current Lola implementation allows to spread its specifications over many specification files. Therefore, a common Lola library with new domain-specific functions can be established.

```
1 macro distance_3D ( double x_1, double y_1, double z_1,
2                     double x_2, double y_2, double z_2 )
3 {
4         sqrt( (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 )
5 }
6
7 input double x,y,z,r,s,t
8 output double point_distance := distance_3d(x,y,z,r,s,t)
```

Listing 15: A Lola specification that computes the distance between two points in a three dimensional space.

Freezing a stream evaluation

Previously, we introduced freezing offsets and windows. Makros can be used to extend this principle to a complete stream evaluation. In Listing 16, we depict the idea. Note that it is possible to encode freezing offsets and windows in this fashion. However, we are restricted by the parameter types. We have to specify a new macro for each possible type which is infeasible having tuples. Unfortunately,

in this example, we suffer from the same restriction. However, in our experience the common types, i.e. integer, double, and string, occur most frequently. A solution might be to remove the obligation of types in the parameters but then static type checking of the macros is not possible anymore.

```
1  macro freezeStreamValue_int (expression then_expr, expression else_expr ,
2                               int freezeTime)
3  {
4      if position % (freezeTime+1) = 0 { then_expr } else { else_expr }
5  }
6  input  double value
7  output double check := freezeStreamValue_int(sqrt(value), check[-1,0.0], 3)
```

Listing 16: A Lola specification that checks whether the sensor is frozen using a normal window.

Abstract Lola specifications

Macros are also very useful to simplify common Lola properties in specifications. For instance, many safety properties involve a bound which should never be exceeded. In Listing 17, we show how to specify bound checks and the common average statistics. In the example, we generate an output stream *avg_value* and an output stream *check_bound* using macros.

```
1  macro boundcheck_double ( stream double s, stream double v, double bound)
2  {   output bool s := v < bound   }
3  macro average_double ( stream double s_out, stream double s_in )
4  {
5      output (int, double) s_out := ( get( s_out[-1,(0,0.0)] , 0) + s_in    ,
6                                      get( s_out , 0 ) / double(position+1) )
7  }
8  input double value
9  boundcheck_double( check_bnd, value, 10.0 )
10 trigger check_bnd
11 average_double( avg_value, value )
```

Listing 17: A Lola specification that uses macros to specify streams.

5.5 Summary

In this section, we introduced the domain-specific Lola extensions, one part of the main contribution of this thesis. We presented the applicable core functions, introduced keywords, and extended possible stream accesses. For the latter we are able to freeze a value for a given time to reduce the computation overhead. Furthermore, we do not only allow to access a relative stream position but, also, an absolute position. Using these features, we avoided auxiliary streams to distinguish between different phases in the trace, e.g. initial values. The resulting specifications are more compact and efficient.

Using the knowledge of domain experts, it is possible to improve the performance of Lola. Prior knowledge over a stream behaviors can be specified. New operators may make use of this to improve the performance and memory consumption. As a first example of such an operator, we presented the switch statement.

In order to improve the usage, specifically the feedback to the user, we extended the online behavior, e.g. triggers, to give feedback at different levels of granularity. Furthermore, we used tagging and filtering to generate a new data set based on the avaible one. Control commands were introduced to express criteria where a Lola monitor should pause, reset, or even exit its evaluation.

In the last section, we proposed macros and gave motivating examples why their usage might be beneficial. A future extension, not mentioned before, are the introduction of sets and lists. In the next section, based on specifications, we see why lists are useful. The idea of sets is to group streams with the same properties together. By unfolding this set, similar the window offset unfolding, we can specify a property once for all streams in the set. The idea is depicted below.

```
1 input int a, b, c
2 const set all_inputs := {a, b, c}
3 output bool bound_check :=  forall i in all_inputs, i < 20
```

6 LOLA in Practice

In this section, we present the application of LOLA in practice by means of experimental results. We use LOLA for both offline and online monitoring. The specifications are constructed based on interviews conducted with the involved working groups. We chose analyzing the system in a bottom-up manner. Detecting erroneous behaviors in lower reasoning stages, e.g. sensor fusion, increases the integrity and robustness of the system. Otherwise, high-level reasoning, e.g. planning, might try to solve these problems due to false correctness assumptions. Considering the LOLA integration into the development stages, we decided to follow the presented stages of verification and validation in Section 3.1.5. In Section 6.1, we show the attained specification. Afterwards, the monitor online integration is presented. In Section 6.3, we explain log file analysis. In Section 6.4, we experiment with SiL simulations and with HiL simulations. Finally, the reoccurring typed of properties are summarized in Section 6.5.

6.1 Specifications

For each group (Section sec:involvedWorkingGroups), we list and explore the attained specifications. In many of them, we see reoccurring properties, e.g. frequency checks. Since these specifications are used in the experiments, we retain them, to give a better impression of the monitoring workload. Keep in mind that the following specifications are monitorable offline as well as online. They will be used to evaluate the impact of the monitor on the overall performance.

Especially, the first specifications are of an analytical nature, where the objective is to determine the boundary values. In order to support this kind of specification, the implementation delivers the last value of each stream at the end of the trace similar to an implicit snapshot. We provide the number of input streams available in the respective log file as comment in the first line of each specification some of them are unused by the specification but passed to the monitor anyways.

Sensor Fusion and Environment Perception

- Magnetometer:
  A sensor has a predefined frequency with which it delivers values. If the sensor is valid, this frequency is fixed and should only vary slightly. Integrating over a too small time interval or a too large interval, can have severe outcomes. In Listing 18, from Line 5 to 6, we compute the frequency, i.e. the difference in time between two consecutive *time* values. Additionally, in Line 4, we calculate the current flight time. Note that *position* is by default monotonically increasing. Therefore, after the second position, we enter the default case only. In order to compute the current time, we receive two time offsets. The input *time_s* is an absolute offset in seconds and *time_ms* is an relative offset on *time_s* in microseconds. Given the frequency, we compute its average in Line 9. We have to account the double occurrence of the first frequency in the devisor (position 0 and 1). We chose this approach because it avoids division by zero which otherwise had to be checked.

  Additionally, in Line 11 to 13, we calculate the maximum jump of the magnetic field vector. Ideally, this maximum should be close to zero. We could define a notification in case of a violation:
  `trigger change_max < 1.0`

```
1  //Amount of input streams in the experiments: 5
2  input double x, y, z, time_s, time_ms
3  output double time := time_s + time_ms / 1000000.0
4  output double flight_time := time - time#[0,0.0]
5  output double frequency := switch position{
6                             case 0  { 1.0 / ( time[1,0.0] - time ) }
7                             default { 1.0 / ( time - time[-1,0.0] ) } }
8  output double freq_sum := freq_sum[-1, 0.0] + frequency
9  output double freq_avg := freq_sum / double(position+1)
10
11 output double euclidian_norm := sqrt( x^2.0 + y^2.0 + z^2.0 )
12 output double euc_change := euclidian_norm - euclidian_norm[-1,0.0]
13 output double change_max := max( euc_change,change_max[-1,double_min] )
```

Listing 18: The specification used for the mgn_output.log.

- Global Positioning System
  Two log files are used to capture the behavior of the GPS sensor: one for the velocity and the other for the position.

  Listing 19 is used for the velocity GPS file. From Line 5 to 11, similar to Listing 18, we compute the average frequency, but, additionally, we capture the maximal and minimal frequency. The next two outputs do the same for vertical and horizontal speed, respectively. The input stream *sol_age* is the differential age of the solution status. Since new values are expected, this value should always be close to zero. In Line 16, we get a notification whenever we enter and leave such a region. The input *trk_gnd_in_bound* checks that the *track over ground*, the motion direction with respect to north, is a valid degree. The values of *trk_gnd_min* and *trk_gnd_max* show only slight deviations when flying in the same direction.

```
1  //Amount of input streams in the experiments: 9
2  input double sol_age, hor_spd, trk_gnd, vert_spd, time_s, time_ms
3  output double time := time_s + time_ms / 1000000.0
4  output double flight_time := time - time#[0,0.0]
5  output double frequency := switch position{
6                              case 0  { 1.0 / ( time[1,0.0] - time ) }
7                              default { 1.0 / ( time - time[-1,0.0] ) } }
8  output double freq_sum := freq_sum[-1, 0.0] + frequency
9  output double freq_avg := freq_sum / double(position+1)
10 output double freq_max := max( frequency, freq_max[-1,double_min] )
11 output double freq_min := min( frequency, freq_min[-1,double_max] )
12
13 output double hor_spd_max  := max(  hor_spd,  hor_spd_max [-1,0.0] )
14 output double vert_spd_max := max(  vert_spd, vert_spd_max[-1,0.0] )
15
16 trigger_change sol_age <= 0.5 with "Sol age should remain zero!"
17 output bool trk_gnd_in_bound := if trk_gnd >= 0.0 & trk_gnd <= 360.0
18                                          { trk_gnd_in_bound[-1,true] }
19                                 else  { false }
20 output double trk_gnd_min := min( trk_gnd, trk_gnd_min[-1,360.0])
21 output double trk_gnd_max := max( trk_gnd, trk_gnd_max[-1,0.0])
```

Listing 19: The specification used for the gps_vel_output.log.

The properties on the GPS position log are specified in Listing 20. Line 13 to 20 records the maximal, and minimal longitude and latitude respectively. A post flight analysis can use these values to detect whether there exist a jump in the GPS signal. Later in this subsection, we specify a property which is able to detect such jumps online. Here, we use these values to check if they comply with their bounds (Line 22 to 25). We do not need the maximal or minimal values to check these bounds. However, this example shows the advantage of the modular composition of LOLA specifications. We capture the boundaries *and* reuse them in other streams which avoids recalculations. In Line 27 to 30, we check the *relative* height behavior and assume a mission bound of 100. Finally, the last properties show that binary encodings of integers are used and that LOLA can handle them. Notice, that in future additional binary transformations are interesting for online monitoring where data is moved across the system. It is cheaper to send such binaries instead of complete objects, e.g. strings.

- Inertial Measurement Unit
  In Listing 21, we show the specification used for the IMU. We assume an *ideal_frequency* of 100Hz. Similar to before, we compute the frequency in each step of the system. However, this time we do not compute the maximum and minimum frequency. Instead, we compute the *deviation* and the *worst deviation* from the ideal frequency in Line 12 and 14. In fact, in Line 14, we also remember the position where it happens[6].

  For analysis purposes, we retain the maximal acceleration in x, y, and z direction. We assume an arbitrarily chosen mission bound of 15. In Line 25 to 30, we check that the acceleration sensors are not frozen by taking a frozen window over the past 6 values. Finally, the *counter* value is used to detect missing IMU values. Ideally, the counter should increase by 1 in each step and reset whenever it reaches 100.

[6] In this manner, we found and displayed a deviation of 11Hz which was by then unknown.

```
//Amount of input streams in the experiments: 18
input double lat, lon, hgt, nObjs, nGPSL1, time_s, time_ms
output double time := time_s + time_ms / 1000000.0
output double flight_time := time - time#[0,0.0]
output double frequency := switch position{
                            case 0  { 1.0 / ( time[1,0.0] - time ) }
                            default { 1.0 / ( time - time[-1,0.0] ) } }
output double freq_sum := freq_sum[-1, 0.0] + frequency
output double freq_avg := freq_sum / double(position+1)
output double freq_max := max( frequency, freq_max[-1, double_min] )
output double freq_min := min( frequency, freq_min[-1, double_max] )

output double lat_max := switch position{
                        case 0 { lat } default{ max(lat, lat_max[-1,0.0]) } }
output double lat_min := switch position{
                        case 0 { lat } default{ min(lat, lat_min[-1,0.0]) } }
output double lon_max := switch position{
                        case 0 { lon } default{ max(lon, lon_max[-1,0.0]) } }
output double lon_min := switch position{
                        case 0 { lon } default{ min(lon, lon_min[-1,0.0]) } }

output bool lat_in_bound := ! (max( abs(lat_max), abs(lat_min) ) <= 90.0)
output bool lon_in_bound := ! (max( abs(lon_max), abs(lon_min) ) <= 180.0)
trigger lat_in_bound with "Irregular latitude value!"
trigger lon_in_bound with "Irregular longitude value!"

output double hgt_inc_max := max( hgt_inc_max[-1,0.0], hgt - hgt#[0,0.0] )
output double hgt_dec_max := min( hgt_dec_max[-1,0.0], hgt - hgt#[0,0.0] )
trigger hgt_inc_max >  100 with "Never increase height by more than 100m!"
trigger hgt_dec_max < -100 with "Never decrease height by more than 100m!"

const int threshold_nObjs  := 10
const int threshold_nGPSL1 := 10
output int nObjs_trust_worthy  := bin_to_int(int(nObjs))
output int nGPSL1_trust_worthy := bin_to_int(int(nGPSL1))
trigger nObjs_trust_worthy  < threshold_nObjs  with "nObjs below threshold!"
trigger nGPSL1_trust_worthy < threshold_nGPSL1 with "nGPSL1 below threshold!"
```

Listing 20: The specification used for the gps_pos_output.log.

```
1  //Amount of input streams in the experiments: 9
2  input double ax, ay, az, time_s, time_ms
3  input int counter
4  const double ideal_frequency := 100.0
5  output double time := time_s + time_ms / 1000000.0
6  output double flight_time := time - time#[0,0.0]
7  output double frequency := switch position{
8                              case 0  { 1.0 / ( time[1,0.0] - time ) }
9                              default { 1.0 / ( time - time[-1,0.0] ) } }
10 output double freq_sum  := freq_sum[-1,0.0] + frequency
11 output double freq_avg  := freq_sum / double(position+1)
12 output double deviation := difference(frequency, ideal_frequency)
13 output bool exceeds_worst     := deviation >  get(worst_dev[-1,(0,0.0)],1)
14 output (int,double) worst_dev := if exceeds_worst { (position,deviation) }
15                                  else { worst_dev[-1,(-1,0.0)] }
16
17 output double ax_max := max( abs(ax), ax_max[-1,0.0] )
18 output double ay_max := max( abs(ay), ay_max[-1,0.0] )
19 output double az_max := max( abs(az), az_max[-1,0.0] )
20 const double a_max := 15.0
21 trigger_change ax  > a_max
22 trigger_change ay  > a_max
23 trigger_change az  > a_max
24
25 output bool frozen_ax := ax[-5..0,0.0,=,6]
26 trigger frozen_ax
27 output bool frozen_ay := ay[-5..0,0.0,=,6]
28 trigger frozen_ay
29 output bool frozen_az := az[-5..0,0.0,=,6]
30 trigger frozen_az
31
32 output bool check_counter := switch position {
33                    case 0 { false }
34                    default { counter != ( (counter[-1, -1] + 1 ) % 100  ) } }
35 trigger  check_counter with "A counter value was ignored."
```

Listing 21: The specification used for the imu_output.log.

- Navigation Output - Result of the Sensor Fusion
  During the discussion of the GPS sensor, we mentioned that we are interested in detecting jumps of sensor values. We use the *nav_output.log* to specify such a property from Line 13 onwards. Given the longitude and latitude, we use the *Haversine formula* [4][7] to estimate the distance between two points on a sphere:

$$a = \sin((\varphi_2 - \varphi_1)/2)^2 + \cos(\varphi_1) * \cos(\varphi_2) * \sin((\lambda_2 - \lambda_1)/2)^2 \quad (1)$$

$$c = 2 * atan2(\sqrt{a}, \sqrt{1-a}) \quad (2)$$

$$d = R * c \quad (3)$$

  where R is the radius of the sphere, here the earth, φ_1 and φ_2 are the latitude and λ_1 and λ_2 are the longitude of the first and the second position, respectively.

  The formula is encoded straight forwardly from Line 22 to 29. Since it expects the latitude and longitude in radians and we receive them in decimal degree we have to convert them first (Line 17 to 20). Again, we see the advantage of the modular composition which allows this conversion to be performed straight forwardly.

  So far, we computed the traveled GPS distance. In Line 32, we calculate the assumed distance given the current velocity and the passed time. By comparing these two values and incorporating a small deviation, we are able to detect a signal jump in Line 35.

Flight Control and Systems Integration
In this paragraph, we are interested in detecting different phases of a flight, e.g. the hover phase. The basic idea is that each possible flight phase has unique properties which distinguish them from others. In practice, it would be great to know these properties. Unfortunately, they are not formally identified, yet. Listing 23 depicts how LOLA can support this task. Tagging can reduce the bulk of data by selecting only the fragments which are interesting. Maybe, the chosen log file does not even contain any interesting fragments. Also, if not yet contained properties are interesting, they can simply be added and e.g. plotted afterwards. In our example, we are interested in fragments of the flight which are longer than five seconds with a *velocity* below 0.3 meter per seconds.

[7] We use a derivation of the original formula which is easier to implement but does not take the ellipsoidal shape of the earth into account.

```
1  //Amount of input streams in the experiments: 34
2  input double lat, lon, ug, vg, wg, time_s, time_ms
3  output double time := time_s + time_ms / 1000000.0
4  output double flight_time := time - time#[0,0.0]
5  output double frequency := switch position{
6                             case 0  { 1.0 / ( time[1,0.0] - time ) }
7                             default { 1.0 / ( time - time[-1,0.0] ) } }
8  output double freq_sum := freq_sum[-1,0.0] + frequency
9  output double freq_avg := freq_sum / double(position+1)
10 output double freq_max := max( frequency, freq_max[-1,double_min] )
11 output double freq_min := min( frequency, freq_min[-1,double_max] )
12
13 output double velocity := sqrt( ug^2.0 + vg^2.0 + wg^2.0 )
14 const double R          := 6373000.0
15 const double pi         := 3.1415926535
16
17 output double lon1_rad := lon[-1,0.0] * pi / 180.0
18 output double lon2_rad := lon * pi / 180.0
19 output double lat1_rad := lat[-1,0.0] * pi / 180.0
20 output double lat2_rad := lat * pi / 180.0
21
22 output double dlon      := lon2_rad - lon1_rad
23 output double dlat      := lat2_rad - lat1_rad
24 output double a := (sin(dlat/2.0))^2.0  +
25                   cos(lat1_rad) *
26                   cos(lat2_rad) *
27                   (sin(dlon/2.0))^2.0
28 output double c := 2.0 * atan2( sqrt(a), sqrt(1.0-a) )
29 output double gps_distance := R * c
30
31 output double passed_time     := time - time[-1,0.0]
32 output double distance_max    := velocity * passed_time
33 output double dif_distance    := gps_distance - distance_max
34 const  double delta_distance  := 1.0
35 output bool   detected_jump   := switch position {
36                                   case 0 { false }
37                                   default { dif_distance >  delta_distance } }
38 snapshot detected_jump with "Invalid GPS signal received!"
```

Listing 22: The specification used for the nav_output.log.

In Line 8, we encode a state machine which remembers whether we begin, we currently are, or we are not, in a phase where the velocity is below the threshold *vel_bound*. The output *start_interval* records the starting time of such a phase. Based on *end_interval*, we write the starting time and the end time into the new log file. It evaluates to *true* whenever we were previously in an interesting phase, we currently reached the end of such a phase, and the phase took longer than the specified time interval of five. This example shows why lists (cf. Section 5) are interesting. Since we are referring to real time, there can be an a priori unbounded amount of data until we reach the time interval. So far, it is not possible to collect these data values and write them to file whenever the tag condition holds. In our example, we remember the start time and end time of the phase and by using them in a consecutive run, we can filter the values of the phase.

Similar, but in an online fashion, we detect phases in Listing 24 from Line 14 to 26. The objective is to identify flight fragments where the velocity differs by only *vel_bound* for at least three seconds. We use the computed *velocity* to shift the maximum and minimum velocity along the trace. But, we reset them whenever they differ by more than the given *vel_bound*. Also, whenever we reset, we record the time which will later be the starting time of such a fragment. The stream *unchanged* triggers a snapshot whenever the fragment exceeds the three seconds bound. In *unchanged*, we count to 150 because we assume an underlying frequency of 50Hz.

An additional interesting property evaluable in this log is the deviation between the actual velocity and the reference velocity, depicted in Line 28 to 35. Ideally, both velocities should not differ significantly. We capture the worst deviation in *worst_dev* as well as the position where it occurs.

In the last lines, we use simple bound checks on the fuel and power, respectively. Note that we notify the user only once whenever a certain threshold is undershot. Also, we use different granularities on the same stream indicating the urgency of the notification.

```
1  //Amount of input streams in the experiments: 97
2  input double time_s, time_ms, vel
3  output double time := time_s + time_ms / 1000000.0
4  const double vel_bound     := 0.3
5  const double time_interval := 5.0
6
7  output bool correct_vel        := abs( vel ) < vel_bound
8  output int cur_state           := if correct_vel {
9                                      if cur_state[-1,0.0] = 0
10                                        { 1 } else { 2 }  }
11                                     else { 0 }
12 output double time_since_start :=  time - start_interval[-1,0.0]
13 output double start_interval   := if cur_state = 2 { start_interval[-1,0.0] }
14                                    else { time }
15 output bool end_interval       := cur_state[-1,0] > 0  & !correct_vel &
16                                    time_since_start > time_interval
17 tag as begin,          end   if  end_interval
18   with start_interval, time  at "Desktop"
```

Listing 23: A simple specification that shows the usage of tagging.

```
1  //Amount of input streams in the experiments: 97
2  input double time_s, time_ms, vel_x, vel_y, vel_z,
3                      fuel, power, vel_r_x, vel_r_y, vel_r_z
4  output double time := time_s + time_ms / 1000000.0
5  output double flight_time := time - time#[0,0.0]
6  output double frequency := switch position{
7                              case 0  { 1.0 / ( time[1,0.0] - time ) }
8                              default { 1.0 / ( time - time[-1,0.0] ) } }
9  output double freq_sum := freq_sum[-1,0.0] + frequency
10 output double freq_avg := freq_sum / double(position+1)
11 output double freq_max := max( frequency, freq_max[-1,double_min] )
12 output double freq_min := min( frequency, freq_min[-1,double_max] )
13
14 const double vel_bound       := 1.0
15 output double velocity       := sqrt( vel_x^2.0 + vel_y^2.0 + vel_z^2.0 )
16 output double velocity_max   := if reset_max[-1,false] { velocity }
17                                 else { max( velocity, velocity_max[-1,0.0]) }
18 output double velocity_min   := if reset_max[-1,false] { velocity }
19                                 else { min( velocity, velocity_min[-1,0.0]) }
20 output double dif_max        := difference(velocity_max, velocity_min)
21 output bool   reset_max      := dif_max > vel_bound
22 output double reset_time     := if reset_max | position = 0 { time }
23                                 else  { reset_time[-1,0.0] }
24 output int unchanged         := if reset_max[-1,false] { 0 }
25                                 else { unchanged[-1,0] + 1 }
26 snapshot unchanged = 150 with "Phase detected!"
27
28 output double  vel_dev := difference(vel_r_x,vel_x) + difference(vel_r_y,vel_y)
29                           + difference(vel_r_z,vel_z)
30 output double  dev_sum   := vel_dev + dev_sum[-1,0]
31 output double  vel_av    := dev_sum / double((position+1)*3)
32 output int worst_dev_pos := if worst_dev[-1,double_min] < vel_dev { position }
33                            else { worst_dev_pos[-1,0] }
34 output double worst_dev  := if worst_dev[-1,double_min] < vel_dev { vel_dev }
35                            else { worst_dev[-1,0.0] }
36
37 output double fuel_p  := ( ( fuel#[0,0.0] - fuel ) /  (fuel#[0,0.0]+0.01)  )
38 output double power_p := ( (power#[0,0.0] - power) /  (power#[0,0.0]+0.01) )
39 trigger_once fuel_p  < 0.50  with "Fuel below half capacity"
40 trigger_once fuel_p  < 0.25  with "Fuel below quarter capacity"
41 trigger_once fuel_p  < 0.10  with "Urgent: Refill Fuel!"
42 trigger_once power_p < 0.50  with "Power below half capacity"
43 trigger_once power_p < 0.25  with "Power below quarter capacity"
44 trigger_once power_p < 0.10  with "Urgent: Recharge Power!"
```

Listing 24: The specification used for the ctrl_output.log.

Mission Planning and Execution
In the following, we take a look at the mission manager which handles the high-level reasoning. The manager is similar to a finite automaton with locations and edges. A location indicates the current state of system and an edge represents the condition and action which leads from one location to another. In general, the current location is written to the log file in each step independently of changes. Filtering the changes by hand can be very exhausting.

By encoding this automaton in LOLA, we can support the debugging process. In Listing 25, we show an example for applying this technique. The objective is to extract a user-friendly interpretation of the walk. We indicate the specification only and leave some automaton parts open. Output *change_state* uses the current location of the automaton and compares it to the previous one. Whenever they differ, we know we identified a taken transition. Instead of reading the enums (ints) of locations, we apply the transition table *state_enum*. Finally, in Line 23, we use the identified location change to construct a string representing the walk. A possible outcome might be: *"Start -> MissionControllerOff -> HammerHeadTurn"* which is an invalid walk.

To detect such invalid walks, we can use the specification in Listing 26 . Basically, we introduce two new streams: *checkCommand_MissionControllerOff* and *no_valid_transition*. Whenever there is a location change, *no_valid_transition* queries the respective helper stream, e.g. *checkCommand_MissionControllerOff*, whether the occurring transition was valid. In case of a violation, the user will receive a snapshot of the current LOLA state. Using this approach, we translate the complete state space into a LOLA specification which is used in the experiments afterwards. It would be possible to incorporate the edges as well, but we omitted this here.

Using the state space, we are not only able to detect invalid transitions, but we can also aggregate statistics for each location, e.g. fuel consumption for each flight mode, or check safety requirements, e.g. landing should take less than ten seconds. If the overall fuel consumption is too high, we can use LOLA to point to the most consuming location. Similar to a program profiler, we should choose this point to apply the first changes. In Listing 27, we show explicit examples for both. From Line 20 to 32, we compute statistics for the *HoverTo* location. An interesting statistic is the duration spend in a specific location. The output stream *entrance_time* takes a timestamp whenever a change in the location occurs. In Line 22, we know that we left the location *HoverTo*. Hence, we compute the time spend in this location and add it to *hover_sum_time* depicted in Line 26. This stream will give us the current time we spend in *HoverTo* up to this point in the trace, i.e. the total time at the end of the trace. Further, we compute the maximal and average time spend in this location which gives us a better understanding of

the location behavior. For instance, do we visit the location several times for a short amount of time or for a few enduringly times only? Note that we do this exemplary for the *HoverTo* locations but we could extend this to other locations as well by reusing stream computations, e.g. *entrance_time*.

As an example for the safety assurance, we chose a time bound on landing. Assume we enter the state *landing*, captured by Line 35. The output stream *landing_info* computes the difference between the entrance of the landing location and the current time. In Line 37, if we still want to land, we are not yet *OnGround*, and the time bound is exceeded, the stream *landing_error* will evaluate to *true*, indicating a violation.

```
1  //Amount of input streams in the experiments: 23
2  input int stateID_SC
3  const int Start                  := 0
4  const int MissionControllerOff   := 1
5                                      ...
6  const int HammerHeadTurn         := 16
7
8  output bool change_state := switch position {
9                                case 0 { false }
10                               default { stateID_SC != stateID_SC[-1,-1] } }
11 trigger change_state
12
13 output string state_enum := switch stateID_SC  {
14                               case 0 { "Start" }
15                               case 1 { "MissionControllerOff" }
16                                ...
17                               case 16{ "HammerHeadTurn" }
18                               default{ "Invalid" }  }
19 output string state_trace :=
20    switch position {  case 0  { state_enum }  default {
21    if change_state { concat(concat(state_trace[-1,""]," -> "),state_enum) }
22    else { state_trace[-1,""] } }  }
```

Listing 25: The specification extracts the state sequence as string, given the missionManager_output.log.

```
1  //Amount of input streams in the experiments: 23
2  input int stateID_SC
3  const int Start                 := 0
4  const int MissionControllerOff  := 1
5                                   ...
6  const int HammerHeadTurn        := 16
7
8  output bool change_state := switch position {
9                              case 0 { false }
10                             default { stateID_SC != stateID_SC[-1,-1] } }
11 trigger change_state
12
13 output bool checkCommand_MissionControllerOff :=
14              switch stateID_SC[-1,-1] {
15               case     Start { true }
16               case     SlowDown { true }
17                ...
18               default         { false }
19             }
20 output bool no_valid_transition := !( change_state -> (position = 0 |
21               switch stateID_SC {
22                case Start                { true }
23                case MissionControllerOff { checkCommand_MissionControllerOff }
24                 ...
25                case HammerHeadTurn       { ... }
26                default { false } } ) )
27 snapshot no_valid_transition
```

Listing 26: The specification checks the state space of the mission manager, given the missionManager_output.log

```
//Amount of input streams in the experiments: 23
input double time_s, time_ms
input int stateID_SC, OnGround
output double time := time_s + time_ms / 1000000.0
output double flight_time := time - time#[0,0.0]
output double frequency := switch position{
                              case 0  { 1.0 / ( time[1,0.0] - time) }
                              default { 1.0 / ( time- time[-1,0.0] ) }
                              }
output double sum_frequency := sum_frequency[-1,0.0] + frequency
output double avg_freq := sum_frequency /  (double(position+1))

const int HoverTo := 4
const int Landing := 5
output bool change_state := switch position {
                              case 0 { false }
                              default { stateID_SC != stateID_SC[-1,-1] } }
trigger change_state

output double entrance_time  := if change_state { time }
                              else { entrance_time[-1,0.0] }
output bool hover_end        := change_state & stateID_SC[-1,-1] = HoverTo
output double hover_cur_time := if hover_end
                                      { time - entrance_time[-1,0.0] }
                              else { 0.0 }
output double hover_sum_time := hover_sum_time[-1,0.0] + hover_cur_time
output int hover_num_times   := hover_num_times[-1,0] +
                                 if hover_end { 1 } else { 0 }
output double hover_max_time := max ( hover_max_time[-1,0.0], hover_cur_time )
output double hover_avg_time := if hover_num_times != 0
                                    { hover_sum_time/double(hover_num_times) }
                              else { 0.0 }

const double landing_timebnd := 20.0
output double landing_info   := if stateID_SC = Landing { 0.0 }
                              else { time - entrance_time[-1,0.0] }
output bool landing_error    := stateID_SC = Landing & OnGround != 1  &
                                 landing_info > landing_timebound
```

Listing 27: A specification that collects properties for the states of the mission manager, given the missionManager_output.log

Another safety assurance, however not concerning the state space, is the adherence of the safety corridor. A mission can consist of several waypoints which have to be passed in a fixed order. The specification given in Listing 28 guarantees the compliance to this order. The streams *at_1*, *at_2*, *at_3*, and *at_4* are used to detect whether we are currently at a waypoint by incorporating a small deviation. In Line 20, *curwp* captures the last seen waypoint and *check_order* compares the changes of *curwp*. To be valid, the waypoint should either remain the same or jump to the next waypoint. Notice that we remain *true* in case of a violation.

So far, in Listing 28, we only guarantee the correct order of the waypoints. In Listing 29, we also check that we do not deviate too much from the next waypoint. We compute the midpoint (Line 16) between the current waypoint (Line 3) and the next waypoint (10). By putting a sphere around this midpoint with the distance to the next waypoint (Line 20 to 24) as radius we have a valid corridor which can guarantee that we are still on track. Additionally, we add a delta to the radius to allow small backwards flights. From Line 25 to 29, we compute the distance between the current position and the center of the current sphere and compare it to the radius. Whenever we leave the sphere, we notify the user. Note that it is possible to put liveness assumption on each waypoint. For instance, we could assure that the time between two consecutive waypoints should never be greater than one minute. To keep the specification concise, we omitted this. The approach is depicted in Figure 8.

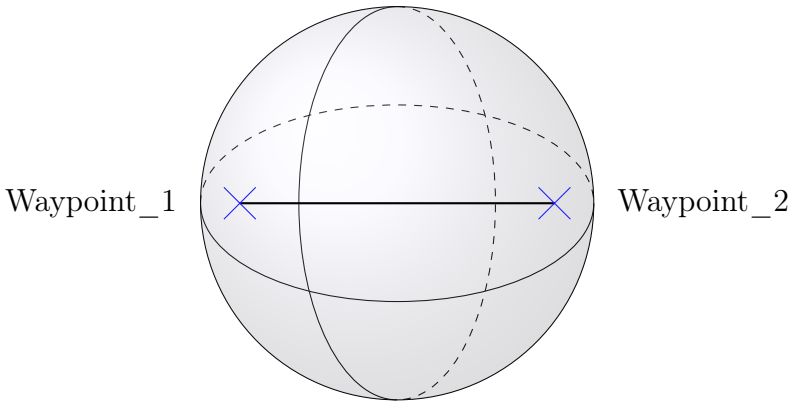


Figure 8: Illustration of a spherical safety corridor used in Listing 29.

```
1  //Amount of input streams in the experiments: 23
2  input double x, y, z
3  const (double, double, double)  waypoint_1 :=  (1.0, 1.0, 1.0)
4  const (double, double, double)  waypoint_2 :=  (2.0, 2.0, 2.0)
5  const (double, double, double)  waypoint_3 :=  (3.0, 3.0, 3.0)
6  const (double, double, double)  waypoint_4 :=  (4.0, 4.0, 4.0)
7  const double delta := 3.0
8  output bool at_1 :=  sqrt( (x - get(waypoint_1, 0))^2.0 +
9                             (y - get(waypoint_1, 1))^2.0 +
10                            (z - get(waypoint_1, 2))^2.0    )  < delta
11 output bool at_2 :=  sqrt( (x - get(waypoint_2, 0))^2.0 +
12                            (y - get(waypoint_2, 1))^2.0 +
13                            (z - get(waypoint_2, 2))^2.0    )  < delta
14 output bool at_3 :=  sqrt( (x - get(waypoint_3, 0))^2.0 +
15                            (y - get(waypoint_3, 1))^2.0 +
16                            (z - get(waypoint_3, 2))^2.0    )  < delta
17 output bool at_4 :=  sqrt( (x - get(waypoint_4, 0))^2.0 +
18                            (y - get(waypoint_4, 1))^2.0 +
19                            (z - get(waypoint_4, 2))^2.0    )  < delta
20 output int curwp :=    if at_1 { 1 }
21                      elif at_2 { 2 }
22                      elif at_3 { 3 }
23                      elif at_4 { 4 }
24                      else { curwp[-1, 0] }
25
26 output bool check_order := check_order[-1,true] &
27                            ( curwp = curwp[-1,0]  |
28                              curwp = curwp[-1,0] + 1 )
29
30 trigger check_order
```

Listing 28: Guarantees on the safety corridors, Version 1.

```
1  //Amount of input streams in the experiments: 23
2  ...addition to Listing 11...
3  output (double,double,double) curwp_loc :=
4          switch curwp {
5           case 0 { (x#[0,0.0], y#[0,0.0], z#[0,0.0]) }
6           case 1 {  waypoint_1 }
7           case 2 {  waypoint_2 }
8           case 3 {  waypoint_3 }
9           default { waypoint_4}    }
10 output (double,double,double) nextwp_loc :=
11         switch curwp + 1 {
12          case 1 {  waypoint_1 }
13          case 2 {  waypoint_2 }
14          case 3 {  waypoint_3 }
15          default { waypoint_4}    }
16 output (double,double,double) center :=
17                          ( (get(nextwp_loc, 0) + get(curwp_loc, 0) ) / 2.0,
18                            (get(nextwp_loc, 1) + get(curwp_loc, 1) ) / 2.0,
19                            (get(nextwp_loc, 2) + get(curwp_loc, 2) ) / 2.0  )
20 output double distance_wps :=  sqrt(
21                            (get(curwp_loc, 0) - get(nextwp_loc, 0))^2.0 +
22                            (get(curwp_loc, 1) - get(nextwp_loc, 1))^2.0 +
23                            (get(curwp_loc, 2) - get(nextwp_loc, 2))^2.0    )
24 output double  radius          :=   distance_wps / 2.0 + 5.0
25 output double distance_center  := sqrt( (x - get(center, 0))^2.0 +
26                                         (y - get(center, 1))^2.0 +
27                                         (z - get(center, 2))^2.0    )
28 output bool    out_corridor    :=  distance_center  > radius
29 trigger_change out_corridor
30 //Possible here to put a time bound on each waypoint
```

Listing 29: Guarantees on the safety corridors, Version 2.

Contingency Manager

In the following, we discuss probabilistic and health reasoning. We assume a contingency manager which takes as input the output of a LOLA monitor. In Listing 30, we show how to model trust based on probabilities. The LOLA monitor receives as input the values of two visual sensors (laser and optical) both arguing over the same observations. The input *avgDist* represents the average distance to the measured obstacles (range of sight), *actual* is the actual number of measurements, and *static* is the number of static measurements, i.e. observations, which remained the same. We model the rating of each of the sensors given arbitrarily chosen weights on the data in Line 3 and Line 6, respectively. The higher the rating the better is the health of the sensor. Using these ratings, we compute the trust on each sensor as a probability and *print* it in each evaluation step. A contingency manager could use this outcome as basis to decide which sensor to use for its reasoning.

```
1  input double avgDist_laser,   actual_laser,   static_laser,
2               avgDist_optical, actual_optical, static_optical
3  output double rating_laser   := 0.2 * static_laser +
4                                  0.4 * actual_laser +
5                                  0.4 * avgDist_laser
6  output double rating_optical := 0.2 * static_optical +
7                                  0.4 * actual_optical +
8                                  0.4 * avgDist_optical
9  output double trust_laser    := rating_laser / ( rating_laser + rating_optical)
10 output double trust_optical  := 1.0 - trust_laser
11 print trust_laser
12 print trust_optical
```

Listing 30: A simple specification that shows how probabilistic reasoning can be used.

Similar in Listing 31, we receive the range of sight of the sensors and the current velocity. In order to guarantee a safe flight, especially under bad weather conditions, it might be beneficial to adapt the velocity to the current visual range. This simple LOLA specification suffices to point to such situations. Based on this, an intelligent adaption of the velocity is possible at runtime.

```
input double avgDist_laser, avgDist_optical, vel
const double vel_warning  := 5.0
const double vel_avoid    := 2.0
output double  avgDst_dif :=  min(avgDist_laser,avgDist_optical) - abs(vel)
trigger avgDst_dif < vel_warning with "WARNING: Dynamic Velocity Limit reached"
trigger avgDst_dif < vel_avoid   with "ERROR: Abort mission."
```

Listing 31: A simple specification that shows how health reasoning can be used.

6.2 Implementation Details

We briefly present some of the C implementation details. Assume the simple specification given below which takes the *s*um over the values v whenever the enable signal e holds. First, the lexer will tokenize the specification and afterwards

```
input  int  v
input  bool e
output int  s   := s[-1,0] + if e { v } else { 0 }
```

the parser returns the abstract syntax trees (AST) for each stream. Figure 9 illustrates the AST corresponding to *sum*.

Next, we compute the time steps in which a value is of interest for the evaluation. In our example, we receive 0 for *v*alue, i.e. only used in this round, and 1 for *s*um, i.e. used in this and in the next round. Further, we compute the size of the ASTs: size(v)=1, size(s)=6. The evaluation is based upon two mayor data

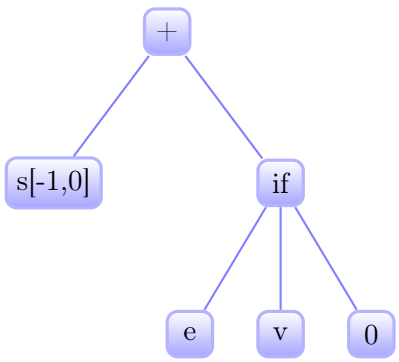


Figure 9: Abstract syntax tree for *sum*.

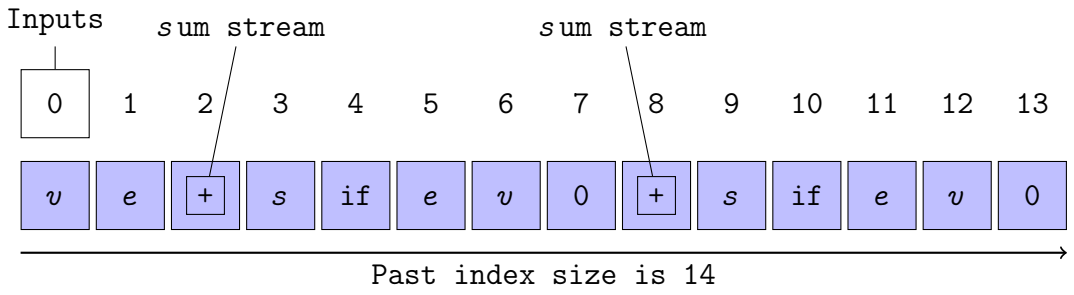


Figure 10: Complete past index.

structures. We call them, the *past index* and the *future index*. The *past index* is allocated once and does not change in size. It contains the previously solved stream instances. For the allocation size, we combine the computed AST sizes and the amount of time they are of interest. The AST for each stream is flattened and pushed into the index. Figure 10 shows this idea for our example specification. First, we flattened and inserted the inputs, followed by the flattened outputs. At Index 2 and 8, we store the *sum* values. By doing so, it is possible to address previous values and automatically remove irrelevant values. For instance, at position 0 of the trace, the current *sum* would be stored at Index 2 and in the next step, the value would be stored at Index 8.

All the calculations are handled due to indexing. Note that the implementation requires the correct evaluation order. We do not check whether a value at a specific index is already calculated or outdated. We omitted this check because the order can be automatically derived based on the dependency graph as indicated in Section 5. This is one of the next implementation steps but for now we focused on the expressiveness and applicability. Also, as a workaround, we allow specifying the evaluation order in the specification itself. If not specified, the implementation assumes the implicit order given by the specification.

The other mayor data structure is called the *future index* which is used to store the streams that could not be evaluate so far. For an efficient lookup, we implemented this index as an inverted index. The mapping takes a key, i.e. a stream value we are waiting for, and returns the waiting streams. Considering the listing below, we obtain the following mapping after the first position:
$a\#1 \rightarrow and\#0$ $b\#1 \rightarrow and\#0$ $and\#0 \rightarrow dis\#0$
We require $a\#1$ and $b\#1$ to solve $and\#0$ which is in return required for $dis\#0$. For efficiency reasons, at the start of a new evaluation round, we insert all the inputs in the next position and then check whether we can resolve some previously unresolved stream values. In our example, we are able to first evaluate $and\#0$ and then using the result, we evaluate $dis\#0$.

Remark: The tuple functions are not yet implemented. For the experiments,

```
1 input  bool a,b,c
2 output bool and :=  a[1,false] & b[1,false]
3 output bool dis :=  c | and
```

we unfold the tuple values. For instance,
`const (double,double,double) waypoint_1 := (1.0,1.0,1.0)` is unfolded to:
`const double waypoint_1_x := 1.0`,
`const double waypoint_1_y := 1.0`, and
`const double waypoint_1_z := 1.0`

To validate the experimental results, we compare the results of this implementation version with our previous one which was not based on indexing only the lexer and parser are used in both.

6.3 Offline Monitoring

In this section, we present the experiments we conducted on the existing log files. First, we explain the experimental setup. The objective is to decide whether LOLA is applicable in the context; detected faults are just a side-effect. In fact, some constants are chosen by plausibility and, thus, may only be false positives. In the following experiments, we use real data of various flights.

6.3.1 Experimental Setup

Offline, LOLA is working as a command-line tool. As parameters, LOLA receives one or more specifications ending with ".lola" and the logged data ending with either ".store" or ".log". In case of several specifications, LOLA automatically combines them into a single one. As an optional parameter, a location for the monitor output can be passed (`-l`). When set, the monitor does not write on the console (default) but writes to the given location instead. An example LOLA call is:
`./lola constants.lola checkHeight.lola heights.store`
For our experiments, we used the specifications presented above and collection of log files obtained from the DLR. We will not elaborate on the log file content here. Further, we wrote a tool which can be set to a root directory and browses through all available subdirectories, looking for a log file which matches a predefined name. To report the memory consumption, we use *time* [11], a unix command-line tool. An example call of our helper tool is:
`/usr/bin/time -lp ./lola spec.lola data.store -l m_log 2 » m_log`
where `m_log` is the monitor log location (optional parameter).

The offline experiments were conducted on a dual-core machine with an 2.6GHz

Intel Core i5 processor with 8GB RAM. The input streams files were stored on an internal SSD.

As an additional feature, the helper tool captures the intermediate results of the LOLA calls and is able to output statistics on *all* of them. For instance, it can compute the average, the maximum, and the minimum frequency deviation over all final LOLA results. This simplifies comparison of several log files allowing to distinguish between fundamental and situational errors. The overall time a specific expert was required to distinguish them was reduced.

The experiments show how LOLA scales regarding runtime and memory for different log files and specifications. As an indicator for the scalability, we state the responsiveness, i.e. the amount of time required for a single event.

6.3.2 Experimental Results

We refer to the specification by the component name and the listing number in brackets. Below, we explain the column abbreviations used for the experimental results.

- #Events ..... *Number of evaluation steps, i.e. rows in log file.*
- FT ..... *Actual flight time.*
- RT ..... *Overall* LOLA *runtime.*
- Responsiveness ..... *Amount of time required per event, i.e.* $\frac{RT}{\#Events}$.
- #Notifications ..... *Total amount of triggered notifications.*
- Memory ..... *Required memory consumption.*

Flight Control and Systems Integration
Specification: CTRL (Listing 24)

#Events	**FT** (sec)	**Responsiveness** (msec)	**#Notifications**	**Memory** (MB)	**RT** (sec)
6275	190	**0,065**	10	1.19	0.41
7409	270	**0,068**	13	1.18	0.51
10516	314	**0,067**	11	1.18	0.71

Sensor Fusion and Environment Perception

- Specification: Magnetometer (Listing 18)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
2137	219	**0,004**	0	0.97	0.01
2640	270	**0,003**	0	0.98	0.01
2887	296	**0,003**	0	0.97	0.01
3110	319	**0,003**	0	0.97	0.01
3129	321	**0,006**	0	0.97	0.02
3428	351	**0,005**	0	0.97	0.02
7307	750	**0,004**	0	0.97	0.03
7490	767	**0,004**	0	0.98	0.03
7754	794	**0,003**	0	0.98	0.03
8739	896	**0,004**	0	0.97	0.04
9086	931	**0,004**	0	0.98	0.04
9636	987	**0,004**	0	0.98	0.04

- Specification: GPS-vel (Listing 19)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
4321	216	**0,006**	154	1.00	0.03
5426	271	**0,007**	246	1.01	0.04
6151	307	**0,008**	52	1.00	0.05
6331	316	**0,006**	222	1.00	0.04
6410	320	**0,007**	317	1.00	0.05
7008	350	**0,007**	348	1.01	0.05
14915	746	**0,007**	564	1.00	0.11
15309	765	**0,007**	124	1.00	0.11
15906	795	**0,008**	224	1.00	0.14
17947	897	**0,007**	682	0.99	0.13
18648	932	**0,007**	88	1.00	0.14
19806	990	**0,007**	24	1.00	0.14

- Specification: GPS-pos (Listing 20)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
4347	217	**0,013**	0	1.05	0.06
5027	251	**0,013**	0	1.06	0.07
6173	309	**0,012**	0	1.04	0.08
6350	317	**0,014**	0	1.04	0.09
6407	320	**0,014**	0	1.04	0.09
7020	351	**0,014**	0	1.04	0.10
14944	747	**0,013**	0	1.04	0.20
15362	768	**0,014**	0	1.04	0.22
15901	795	**0,017**	0	1.04	0.28
17960	898	**0,014**	0	1.04	0.26
18622	931	**0,014**	0	1.04	0.27
19812	991	**0,013**	0	1.06	0.27

- Specification: IMU (Listing 21)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
21570	216	**0,007**	0	1.02	0.17
27391	274	**0,007**	0	1.02	0.21
30454	305	**0,008**	0	1.02	0.25
31998	320	**0,008**	0	1.04	0.26
32235	322	**0,007**	0	1.04	0.24
35013	350	**0,007**	6	1.03	0.28
74676	747	**0,007**	0	1.04	0.59
76494	765	**0,007**	0	1.02	0.60
79236	792	**0,007**	0	1.03	0.63
89800	898	**0,008**	1183	1.03	0.74
93228	932	**0,008**	0	1.04	0.78
98916	989	**0,008**	21	1.05	0.80

- Specification: Navigation output (Listing 22)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
2564	26	**0,027**	0	1.08	0.07
21557	216	**0,025**	0	1.08	0.55
27307	273	**0,025**	0	1.07	0.70
29657	297	**0,025**	0	1.09	0.77
30477	305	**0,025**	0	1.07	0.77
31903	319	**0,024**	0	1.07	0.78
34594	346	**0,025**	0	1.08	0.88
73956	740	**0,024**	0	1.09	1.78
74884	749	**0,024**	0	1.07	1.86
76640	766	**0,024**	0	1.09	1.85
89803	898	**0,025**	0	1.08	2.32
92796	928	**0,024**	0	1.09	2.26
99506	995	**0,024**	0	1.09	2.44

Mission Planning and Execution
Specification: Mission Manager (Listing 25, 26, and 27 as one specification)

#Events	FT (sec)	Responsiveness (msec)	#Notifications	Memory (MB)	RT (sec)
11045	221	**0,017**	11	1.20	0.19
13702	274	**0,018**	4	1.18	0.25
15319	306	**0,018**	15	1.19	0.28
15691	314	**0,017**	15	1.20	0.28
15835	322	**0,017**	26	1.21	0.28
17460	349	**0,017**	0	1.18	0.31
37235	745	**0,019**	69	1.24	0.68
38342	767	**0,018**	61	1.21	0.71
39852	797	**0,018**	68	1.23	0.72
44962	899	**0,017**	8	1.20	0.78
46940	939	**0,018**	73	1.23	0.87
49573	991	**0,018**	69	1.25	0.92

6.3.3 Analysis and Evaluation

We start by discussing the obtained notifications component-by-component. Afterwards, we analyze and evaluate the overall experimental results. Note that we report the total number of notifications, i.e. the sum over all occurred notifications.

Flight Control and Systems Integration
We consider three flights because we only evaluate those flights that match with our stated CTRL specification in Listing 24, i.e. the number of inputs. In this specification, we start a phase based on a defined velocity bound and a minimum phase duration (Line 14 to 26). In all the experiments, we were able to identify at least ten phases.

Sensor Fusion and Environment Perception
The monitor for the magnetometer logs did not raise any notification. This is simply the case because we did not specify any observable, e.g. a trigger.

In contrast, several notifications were raised in the GPS-vel logs. In Listing 19, we specify a *trigger_change* on the *sol_age*. Therefore, we know that there exist several segments in the log where the age of the solution status is greater than 0.5.

The specified bound checks for the GPS-pos log (Listing 20), e.g. latitude ranges from -90 to 90, are never triggered.

In the used specification for the IMU (Listing 21), we specify triggers for the maximal acceleration (Line 21 to 23) and check that the values change within six evaluation steps (Line 25 to 30). In the results, the peak of 1183 notifications stands out. By examining the monitor output and validating the findings for the respective log, we found out that all notifications are solely based on frozen values. The same holds for the other notifications of this component.

In the specification for the navigation log (Listing 22), we identify GPS signal jumps. The monitor result shows that such signal jumps did not occur in any examined flights.

Mission Planning and Execution
The last specification we consider is used for the mission manager where we validate the state chart sequence. In Listing 25, we specify a trigger whenever the state changes. Further, in Listing 26, we use a trigger to identify invalid state transitions. The monitor results show that both triggers occur. By checking the final *state_enum* (Listing 25, the string representation of the state sequence), we can easily identify that *invalid* states were reached. We could also find this invalid states in the respective log data. Nevertheless, we mainly use flight data from 2014 and, therefore, it is more likely that a different state chart was used than the occurrence of an actual violation.

Overall Experimental Results
The examined flight times range from thirty seconds to twenty minutes. The number of events depend on the frequency of the corresponding component. For instance, the IMU ideally runs with 100Hz. This is perfectly reflected in the IMU results by comparing the number of events with the flight time (rounded).

The main result of these experiments is that throughout all experiments, we require constant memory and linear runtime. By responsiveness (in msec), i.e. the time required for a single event, we can validate that the runtime is independent of the length of the trace (#Events). Here, we als require constant memory and the computational overhead is hardly measurable.

Considering the mission manager, we require constant memory. However, in general, this is not the case. Since we accumulate the visited states in a string, the required memory can no longer be bounded to a constant and, thus, is no longer efficiently monitorable in the worst case.

Overall, the efficient memory consumption and very fast responsiveness (hardly measurable) are promising results for the subsequent subsection, where we apply online monitoring. The aim of further experiments is to evaluate the impact of monitoring to the overall system.

6.4 Online Monitoring

In the previous experiments, we have considered the runtime and memory consumption of Lola for used specifications. There, the data was completely available. In this section, we receive the data successively. Since we use efficiently monitorable specifications only[8] this may alter the runtime[9] but not the memory complexity.

In the next subsections, we present the online integration and the experimental setups for both SiL and HiL. Afterwards, we analyze and discuss the experimental results.

In Section 2, we mentioned the notion of *unobtrusiveness*, i.e. the monitor should not alter the critical properties of the system. We highlight some ARTIS architectural limitations to motivate our design choices and indicate why unobtrusiveness may suffer. One restriction is the absence of both a software bus and a hardware bus. Thus, we have to weave a monitor interface into the existing system to supply the monitor with data. Further, for instance, HiL allows arbitrary switching between logging and non-logging. Since the monitor interface is weaved into the logging code segments, we offer the same possibilities for monitoring. The

[8] As mentioned, the mission manager is in fact not efficiently monitorable. However, in our experiments, only a small amount of state changes occur.

[9] The monitor may wait until it receives the next values.

log files are particularly interesting as we can use them to evaluate the monitor outcome and impact on the system by an offline LOLA analysis.

6.4.1 Online Integration

The LOLA online interface is written in C++ and integrated into an ARTIS green version branch. The monitors are implemented as threads which allows spawning a thread whenever logging is turned on and, respectively, its termination whenever it is turned off. The monitor threads belong to the system which is not optimal for unobtrusiveness but facilitates first experimental results. In fact, the experiments will show that the system remains unaffected by the monitoring in our settings. Additionally, the interface can be easily extended to decouple monitoring from the system, e.g. as a process. Listing 32 depicts the available function calls. We consider Listing 33 to explain their usage.

```
1 int init();
2 int run();
3 int eval();
4 int insert_value( streamname, streamtype, value, evalstyle=AUTO );
5 int insert_values( streamnames, streamtypes, list of values, evalstyle=AUTO );
6 int synchronize_value( streamname, streamtype, value, evalstyle=AUTO );
7 int listen( streamname, callbackFunction );
8 int terminate();
```

Listing 32: The functions of the online monitor interface are depicted.

In Line 4, when starting the logging, we create the monitor by passing a specification file, an existing stream in the specification, and a location where the monitor should write its output (by default: stdout). Next, we call `init` which represents the initialization of the monitor, i.e. lexing, parsing, and creating data structures. Only after the thread is ready, this function returns. Afterwards, in Line 6, we set the monitor to `run`, indicating that the monitor is ready to receive values.

In our example, in Line 12, we use `insert_value` to pass the data to the monitor. We have to specify the stream (`"vel"`) to which the value belongs and its type. The flag `MANUAL` indicates that the monitor should wait for an `eval` call which represents the start of a LOLA evaluation phase. The evaluation phase covers all currently possible evaluation steps, i.e. the received values are buffered. This allows the user to enforce that the LOLA monitor evaluates ten evaluation steps at a time instead of only single ones. As an alternative, `AUTO` is offered which

automatically determines whether all streams have an unevaluated value. Another way to add several values to possibly several streams is `insert_values`).

Somewhat different is the function `synchronize_value` which also adds a value to a stream. The idea is that some streams receive values faster than others and since LOLA assumes a synchronous system, we have to map multiple values of the faster stream to a single value of slower streams. The clock used to synchronize these stream values is passed to the constructor of the monitor, e.g. *time_s* in Line 4. By now, the current implementation takes only the most recent value, but in future, functions over these values are possible, e.g. average.

Finally, in Line 17, we `terminate` the monitor which again returns after completion since we want to allow to restart another one. The last function to mention is `listen` which is designed to accept stream observers. Unfortunately, this function is not yet ready to use. Its idea is to register observers on streams. Whenever a stream value is generated, it is propagated to all its observers by calling their `callbackFunction`. A possible application is the *online visualization* of streams with a python plotting script as observer.

```
1  #include "monitorInterface.hpp"
2  int startLoggingMM(){
3      ...
4      Monitor* monitor = new Monitor(specification, "time_s", monitorlog)
5      monitor->init();
6      monitor->run();
7  }
8  int logData(){
9      double vel_data = getVelocity();
10     ...
11     if(monitor != NULL){
12         monitor->insert_value("vel", DOUBLE, vel_data, MANUAL);
13         monitor->eval();
14     }
15 }
16 int stopLoggingMM(){
17     monitor->terminate();
18 }
```

Listing 33: A simple example how the monitor is weaved into the logging of the system.

In the next subsection, the implementation*s* of the inter-thread communication is explained since we had to adapt to compiler and system constraints.

6.4.2 Experimental Setup

Regarding the experimental setup, we distinguish between the software-in-the-loop and hardware-in-the-loop setup.

Software-in-the-loop Setup
The SiL experiments were conducted on a quad-core machine with an 2.8GHz Intel Core Q9550 processor with 4GB RAM. As operating system, Windows 7 is installed. For the inter-thread communication, to deliver data, we use Windows named pipes[10] to pass the data and callback functions to deliver the monitor outcome.

As simulations, we choose *MissionPlannerPseudoOnlineBasic* and *MissionplannerOnline*. Both use online planning to avoid unknown obstacles in real time. Since we are considering SiL simulations, ideal sensor values are provided.

However, in *MissionPlannerPseudoOnlineBasic*, the obstacle recognition, planning, and control is done synchronously. In *MissionplannerOnline*, the planning is an external process. Therefore, decisions of the planner have to be in time and too late decisions are considered as critical for the system. In the experiments, this constraint is represented by #Below10Hz. Additionally, if previous decisions aggregate over time, the planner should replan the previously determined flight path.

The *MissionPlannerPseudoOnlineBasic* simulation generates logs for the mission manager, the navigation, and the control. Since our monitor interface is integrated at these position, we can monitor them. Except for the control log, the *MissionplannerOnline* generates the same logs. The CTRL log is restricted to seven columns and, therefore, the used specification (Listing 24) can only evaluate the minimum, maximum, and average frequency (Line 4 to 12). The other specifications remain unchanged.

We conducted the experiments to evaluate the impact of monitoring to the runtime behavior on SiL simulations of the system. For the evaluation, we used the computed average frequency and the actual flight time by the monitor, the required time for the whole test, the overhead, and the time required for the same offline analysis. The computed average frequency and the flight time should remain the same. In order to compute the overhead, i.e. the additional time due to monitoring, we compare the test time with monitoring enabled/disabled. Ideally, a low overhead is achieved. Furthermore, for *MissionPlannerPseudoOnlineBasic*, we report

[10] Pipes offer an easy way to buffer the incoming values.

the offline monitoring time to indicate the best case monitor runtime, i.e. without waiting time for the next stream values. Instead, for *MissionplannerOnline*, we depict the amount of times the planner violates its predefined frequency of 10Hz. In the best case, no violations occur. In all experiments, with and without monitor, all unit tests for the simulation are passed. For the reference values, i.e. without a monitor, we apply offline monitoring.

The results are depicted in Section 6.4.3 and analyzed in Section 6.4.5.

Hardware-in-the-loop Setup

The HiL experiments were run on an Intel Pentium with a 1.8GHz and 1GB RAM. The embedded-pc runs a unix system. As inter-thread communication, we use shared memory for both the stream value delivery and the monitor output. To simulate a pipe, we use a mutex protected list[11].

For the HiL simulation, a world and a flight mission was chosen. The world *Hillerse_ v2.1* and two different missions were used: *Hillerse Standard-Testflug HV* and *Hillerse Standard-Testflug FT*. The worlds and the missions are depicted in Figure 11 and 12. In both missions, a set of ordered waypoints was given and during the planned flight, each waypoint had to be visited in a particular order.

In the experiments, we monitored the following components with their specifications in brackets: flight control (Listing 24), navigation (Listing 22), and mission manager (Listing25, 26, and 27 as one specification). Considering the flight control specification, we evaluate all properties but transmit the relevant inputs to the monitor.

For each experiment, first, we reset the dSPACE (the environment simulation)[2] and select the world and the mission. Second, we initialize the navigation values and manually start the logging/monitoring. Then, the mission is planned, uploaded, and started. Finally, after the last mission waypoint is reached, we stop the logging/monitoring and the mission.

The impact of monitoring on the HiL simulation is reflected by the following experiments. For each mission, all experiments flew the same planned route without noticeable deviations. To validate the deviation, an online visualization of the flight was observed. In order to indicate the impact of monitoring, we use the average frequency computed by the monitor, if available, otherwise the average frequency is computed due to offline monitoring on the logged flights. Ideally, the frequency flow should remain unchanged, whether we use monitoring or not.

The results are depicted in Section 6.4.4 and analyzed in Section 6.4.5.

[11] The monitor interface implementation can handle unix pipes but we ran into problems with the embedded-pc pipes.

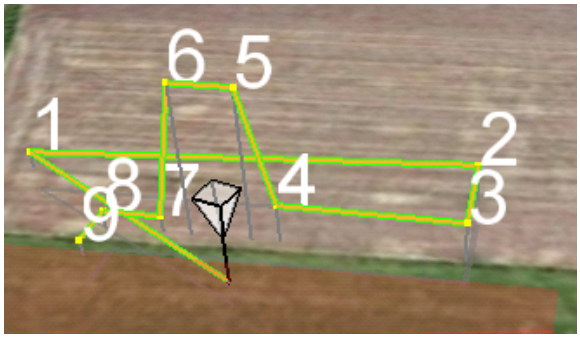


Figure 11: The *Hillerse Standard-Testflug HV* is depicted. Each waypoint shall be visited in order. Between each waypoint, the UAV hovers, to orient itself towards the next waypoint. The UAV is currently at the first waypoint.

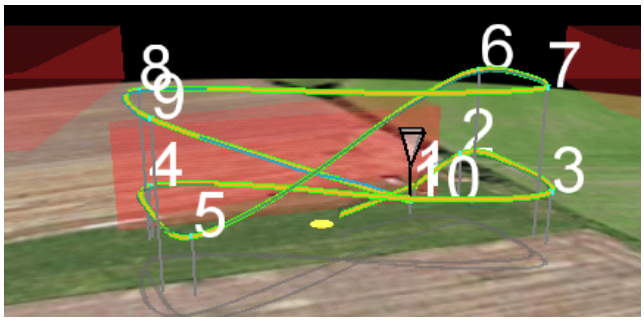


Figure 12: The *Hillerse Standard-Testflug FT* is depicted. Each waypoint shall be visited in order. A spline is flown to visit them. The UAV finished the mission and is currently at the last waypoint. The blue line is the actual flown path, with monitoring.

6.4.3 Experimental Results - Software-in-the-loop Simulation

- Threads ..... *Enumerates the running threads.*
- EvalStep ..... *Amount of values until evaluation step.*
- AvgFreq ..... *Average frequency, computed by monitor.*
- FT ..... *Actual flight time.*
- TestT ..... *Total jUnit test time.*
- Overhead ..... *Additional time due to monitoring, i.e.* $\frac{TestT - Reference\,TestT}{Reference\,TestT}$.
- OfflineT ..... *Offline* LOLA *runtime on the created log file.*
- #Below10Hz ..... *Amount of times the planner frequency was below 10Hz.*

We analyze the results in Section 6.4.5.

Simulation: MissionPlannerPseudoOnlineBasic

Threads	**EvalStep**	**AvgFreq** (Hz)	**FT** (sec)	**TestT** (sec)	**Overhead** (%)	**OfflineT** (sec)
Reference:	-	*ctrl: 50.0*	*107.2*	*526.3*	*0*	*0.06*
No Monitor	-	*mgr: 50.0*	*107.2*	"	"	*0.12*
used	-	*nav: 50.0*	*107.1*	"	"	*0.14*
ctrl_monitor	1	ctrl: 50.0	107.2	523.4	**-0.006**	0.06
mgr_monitor	1	ctrl: 50.0	107.2	524.1	**-0.004**	0.13
nav_monitor	1	nav: 50.0	107.1	528.5	**0.004**	0.15
ctrl_monitor	1	ctrl: 50.0	107.2	528.8	**0.005**	0.07
nav_monitor	1	nav: 50.0	107.1	"	"	0.15
ctrl_monitor	1	ctrl: 50.0	107.2	530.6	**0.008**	0.06
mgr_monitor	1	nav: 50.0	107.2	"	"	0.12
nav_monitor	1	mgr: 50.0	107.1	"	"	0.15
ctrl_monitor	10	ctrl: 50.0	107.1	530.0	**0.007**	0.05
mgr_monitor	10	nav: 50.0	107.1	"	"	0.11
nav_monitor	10	mgr: 50.0	107.1	"	"	0.16
ctrl_monitor	100	nav: 50.0	107.1	524.5	**-0.003**	0.06
nav_monitor	100	ctrl: 50.0	107.1	"	"	0.14
ctrl_monitor	100	ctrl: 50.0	107.1	530.0	**0.007**	0.05
mgr_monitor	100	nav: 50.0	107.1	"	"	0.11
nav_monitor	100	mgr: 50.0	107.1	"	"	0.14
ctrl_monitor	1000	ctrl: 50.0	107.1	530.2	**0.007**	0.05
mgr_monitor	1000	nav: 50.0	107.1	"	"	0.11
nav_monitor	1000	mgr: 50.0	107.1	"	"	0.14

Simulation: MissionplannerOnline

Threads	**EvalStep**	**AvgFreq** (Hz)	**FT** (sec)	**TestT** (sec)	**Overhead** (%)	**#Below10Hz**
Reference:	-	*mgr: 51.0*	*195.8*	*232.6*	*0*	*130*
No Monitor	-	*nav: 50.0*	*195.8*	"	"	"
mgr_monitor	1	mgr: 50.3	196.0	232.4	**-0,001**	132
mgr_monitor	1	mgr: 50.4	194.7	232.6	**0**	145
nav_monitor	1	nav: 50.0	194.8	"	"	"
mgr_monitor	10	mgr: 50.4	197.0	234.6	**0,009**	124
nav_monitor	10	nav: 50.0	197.2	"	"	"
mgr_monitor	100	mgr: 50.5	194.4	230.4	**-0,009**	135
nav_monitor	100	nav: 50.0	194.5	"	"	"
mgr_monitor	1000	mgr: 51.3	195.9	232.8	**0,001**	134
nav_monitor	1000	nav: 50.0	196.0	"	"	"

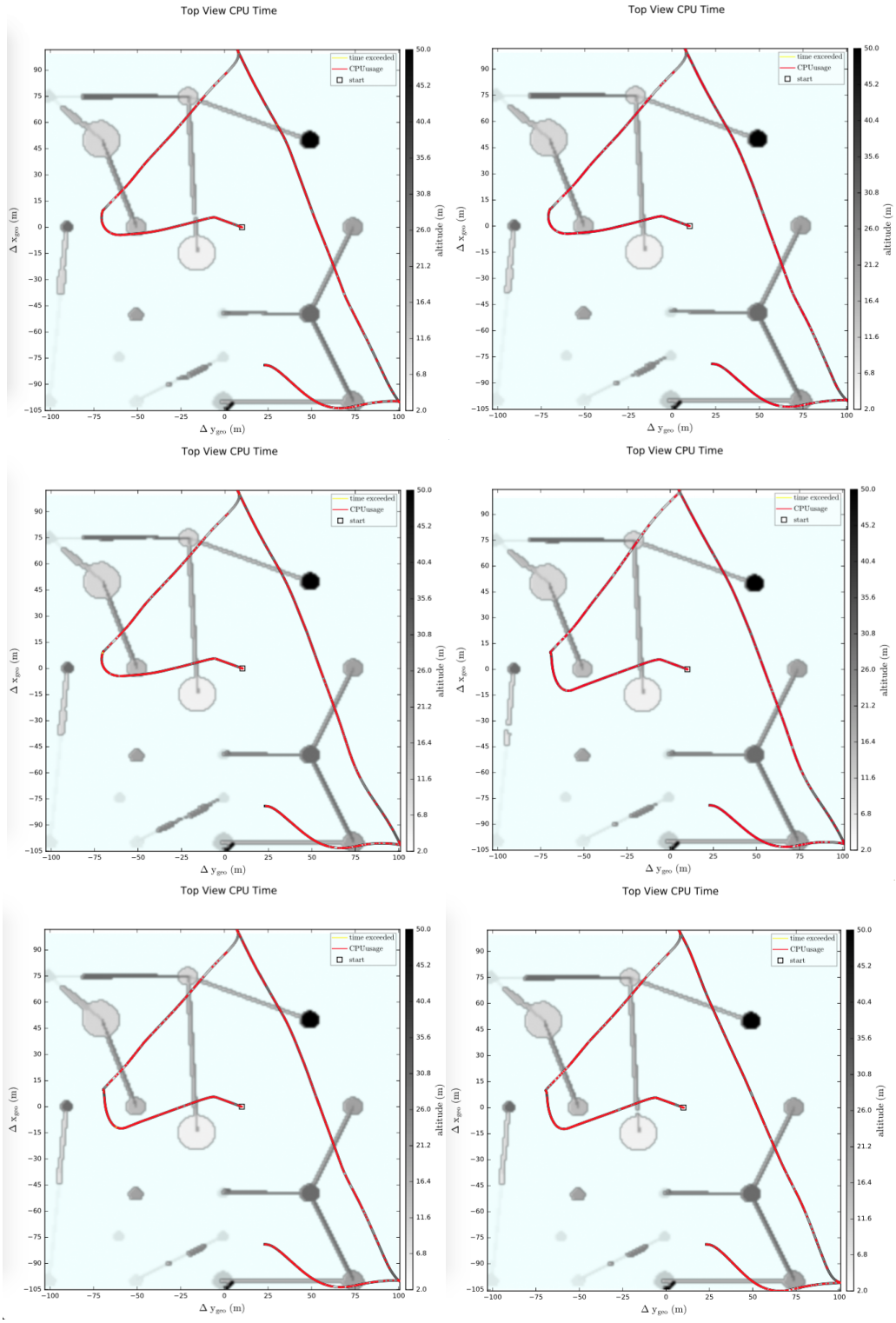


Figure 13: Python plots for the flight path in the software-in-the-loop simulation *MissionplannerOnline* in Section 6.4. The plots are ordered by the experiment result table. The top left plot depicts the simulation only with *mgr_monitor* running and bottom right depicts the simulation with *mgr_monitor* and *nav_monitor* running with *EvalStep* 1000.

6.4.4 Experimental Results - Hardware-in-the-loop Simulation

– Threads (EvalStep) ... *Enumerates the running threads and their eval-step.*
– Online - AvgFreq ..... *Average frequency, bold if computed by the monitor.*
– Offline - AvgFreq ..... *Offline computed average frequency.*

We use the respective specification for the offline analysis of the logged flight. The experimental results are analyzed in Section 6.4.5.

Mission: Hillerse Standard-Testflug HV

Threads (EvalStep)	**Online** - AvgFreq			**Offline** - AvgFreq			
	nav (Hz)	ctrl (Hz)	mgr (Hz)	gps-p (Hz)	gps-v (Hz)	imu (Hz)	mgn (Hz)
No Monitor	*50.0*	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor (1)	**50.0**	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor (1)	**50.0**	-	*50.0*	20.0	20.0	100.0	10.0
ctrl_monitor (1)	-	**50.0**	-	"	"	"	"
nav_monitor (1)	**50.0**	-	-	20.0	20.0	100.0	10.0
ctrl_monitor (1)	-	**50.0**	-	"	"	"	"
mgr_monitor (1)	-	-	**50.0**	"	"	"	"
nav_monitor (100)	**50.1**	-	-	20.0	20.0	100.1	10.0
ctrl_monitor (100)	-	**50.3**	-	"	"	"	"
mgr_monitor (100)	-	-	**50.2**	"	"	"	"

Mission: Hillerse Standard-Testflug FT

Threads (EvalStep)	**Online** - AvgFreq			**Offline** - AvgFreq			
	nav (Hz)	ctrl (Hz)	mgr (Hz)	gps-p (Hz)	gps-v (Hz)	imu (Hz)	mgn (Hz)
No Monitor	*50.0*	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor (1)	**50.0**	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor (1)	**50.0**	-	*50.0*	20.0	20.0	100.0	10.0
ctrl_monitor (1)	-	**50.0**	-	"	"	"	"
nav_monitor (1)	**50.0**	-	-	20.0	20.0	100.0	10.0
ctrl_monitor (1)	-	**50.0**	-	"	"	"	"
mgr_monitor (1)	-	-	**50.0**	"	"	"	"
nav_monitor (100)	**50.0**	-	-	20.0	20.0	100.1	10.0
ctrl_monitor (100)	-	**50.2**	-	"	"	"	"
mgr_monitor (100)	-	-	**50.2**	"	"	"	"

6.4.5 Analysis and Evaluation

First, we analyze the software-in-the-loop experimental results. Afterwards, the hardware-in-the-loop simulation results are considered. Finally, we analyze and evaluate experimental results in general.

Software-in-the-loop Simulation
In Section 6.4.3, different variations of monitor threads with various evaluation steps are used to evaluate the applicability of monitoring for SiL simulations. *EvalStep 1* represents that as soon as all streams received a single value, a Lola evaluation step is executed. Whereas, *EvalStep 100* represents that Lola starts 100 consecutive evaluation steps whenever each stream holds 100 values. Therefore, the responsiveness[12] may suffer but in case Lola causes frequency violations, the number of violations could be reduced.

Considering the *MissionPlannerPseudoOnlineBasic* simulation, no deviations from the average frequencies are detected and the flight time remains the same. The required time to analyze the produced log files indicates that the flight is similar to the flights in Section 6.3.

Considering the *MissionplannerOnline* simulation, slight deviations in the average frequencies are observable. Especially, increasing *EvalStep* leads to higher average frequencies. The actual flight times slightly vary but, as Figure 13 shows, the paths remain unchanged.

The amount of *below 10Hz* violations (#Below10Hz) indicate that adjusting the *EvalStep* might help to reduce the amount of violations. A step size of 10 seems to be reasonable. However, in order to confirm this observation, additional experiments are required.

Further, for both SiL simulations, the overhead, i.e. running the simulation with monitoring, is below one percent and, thus, hardly measurable.

Hardware-in-the-loop Simulation
In Section 6.4.4, different settings of monitor threads with various evaluation steps are examined. The *EvalStep* is given in brackets next to the threads. To decide the impact of monitoring, the frequencies of the components were evaluated and the path of the UAV is manually observed. Throughout all HiL experiments with *EvalStep* 1, the frequencies remain unchanged compared to not using a monitor.

Using an *EvalStep* of 100, the frequencies minimally vary and in fact influence the IMU frequency. Further experiments would be interesting, to reproduce this observation.

[12] Violations are only detected after the required values are passed to the streams.

Experimental Results in General
In SiL, the overhead is very small and the frequencies are unchanged. Frequencies which deviate too much from the expected would directly impact the behavior of the vehicle. In HiL, we focused solely on the frequency violations which mostly remain unchanged. There, it is not possible to compute the overhead because we manually start a mission and end the mission and, thus, we cannot guarantee the same initial conditions. Further, we can only observe the online visualization to recognize path deviations.

However, overall we can conclude that LOLA can be used for both SiL and HiL simulations without significantly affecting the system. We showed that the frequencies are unaffected and depicted the flight path in SiL to show that no changes occurred due to the mission planner.

6.5 Family of Specifications

Concluding this section, we identify common types of properties in the specifications. All of them were covered in Section 6.1. In Section 6.3 and 6.4, the experiments show that most of them are applicable in practice.

- Bound Checks `a < 10`
- Range Checks `a < 10 & a > 0`
- Cross Checks `difference(a,b) < delta`
- Sanity Checks `counter[-1,0] + 1 = counter`
- Data Analysis `max(vel,vel_max[-1,0])`
- Statistics Average frequency
- Plausibility Checks Signal jumps
- Pattern Recognition Phase detection
- Probabilistic Reasoning Sensor trust
- Health Reasoning Decision-making

7 Conclusion

In this section, we provide a summary of the presented work and address some future work. We consider possible extensions to the experimental setup, how to improve unobtrusiveness, what extensions to the monitoring interface and to the specification language Lola are possible, and how to encourage the usage in practice.

7.1 Summary

Runtime monitoring is a formal method for checking whether an execution of a system satisfies a set of desirable properties. The properties can either be checked at runtime or after the system has terminated by analyzing log files. A specification language is used to formalize the properties and provides mathematical guarantees on the reliability of the system.

In this thesis, we considered the stream-based specification language Lola. The general idea of Lola is to generate a set of *output* streams based on a given set of *input* streams. Streams are typed sequences of data. Output stream are defined by an expression over input streams, constants, and other output streams which allows to establish a correlation between several stream. The key feature of Lola is that previous, present, and future stream values can be accessed and included in calculations which enables to express temporal properties and incrementally computable statistical measures. With Lola, we can *formally define* temporal properties like *after receiving a landing command, the height should decrease and the ground should be reached within ten seconds* or statistical measures like *the average frequency* and *the maximum deviation from the ideal frequency during the execution.*

The goal of this thesis was to elaborate the applicability of Lola in the context of unmanned aircraft. The research was carried out in collaboration with the German Aerospace Center (DLR). In Section 3.1, we presented the DLR and indicated the important aspects of autonomy concepts such as correctness, safety, and system health management. Possible applications of runtime monitoring were presented to support the implementation of these aspects. For instance, monitoring can be used to increase the situational awareness of the system state. By identifying unintended (incorrect, unsafe) system states, notifications for violations can be raised. Furthermore, we can use monitoring to reason about the error source, given the identified system faults. By detecting the error source, a contingency manager can initiate countermeasures to mitigate the error impact and, therefore, improves the system health.

In practice, specification languages are often designed for a specific domain to be able to express complex but essential properties of the domain in an under-

standable and still efficient way. We examined the applicability of LOLA based on discussions with DLR engineers. Involved working groups were: Sensor Fusion and Environment Perception, Flight Control and Systems Integration, and Mission Planning and Execution. We were able to collect and formalize multiple properties in LOLA. Further, we chose to analyze the system in a bottom-up manner from sensors validation to path execution. Reoccurring types of properties that occurred in various specifications were identified. Those typed include simple bound, range, cross and sanity checks along with more involved properties like statistical measurements, anomaly detection, and even health reasoning.

To adapt to the domain requirements, new LOLA operators were introduced in order to keep the specification concise and to increase the efficiency. In Section 5, we introduced keywords, freezing of stream values, and absolute stream access. We were able to express generic specifications that do not require adjustment to constant values for each individual flight, such as the initial height.

A further extension is the incorporation of prior domain knowledge, e.g. a stream is monotonically increasing. Knowing that some stream values can never reappear is the key idea of the *switch* operator introduced in Section 5.2. We were able to take advantage of this knowledge and could avoid updating irrelevant parts of the computations in specifications, increasing the monitor efficiency. To improve the user interaction, we extended online feedback, introduced offline feedback, and allowed directly controlling LOLA. Considering online feedback, we facilitate user notifications at different granularity levels. For instance, using `print s` on a boolean stream `s` provides the user with each evaluated stream value whereas `trigger_once s` provides only the value and the position where `s` holds for the first time. Offline feedback was used to generate new data logs based on currently evaluating data. Therefore, a post flight analysis can be carried out on enriched or filtered data logs. With filtering, we can narrow down the faults for properties that are not identified yet and, therefore, cannot be expressed in a LOLA specification. Enriching data logs simplifies creating more expressive plots supporting the reasoning process of engineers. Controls allow a user to `exit`, `reset`, or `pause` the current LOLA evaluation. This is particular useful as early test termination can be based on more sophisticated LOLA conditions and online monitoring can be paused, e.g. when a countermeasure to *abort the mission* is initiated or whenever monitoring affects the system significantly.

Our experiments on offline monitoring showed that LOLA is able to evaluate streams both fast and memory efficient. This behavior carried over to online monitoring where the overall system was almost unaffected by the additional workload of monitoring. To achieve a better understanding of these results, we presented some implementation details (cf. Section 6.2). Two main data structures were used for the evaluation: the past index and the future index. The past index handles all

evaluated stream values and the future index contains the values that are yet to be evaluated. For both online and offline experiments, we used different monitor settings along with the specification presented in Section 6.1 which entail most of the desirable properties. For online monitoring, the monitors are weaved into the system as threads since neither hardware nor software bus are currently available. Further, we deployed Lola in software-in-the-loop and hardware-in-the-loop simulations where all tests were passed without any deviations of the flight paths. Lola has proven to be very well suited to be applied in the field unmanned aircraft for post-flight analysis and simulation purposes.

7.2 Future Work

In the following, to see Lola flying someday, we discuss some aspects that require further research and provide a basis for future work.

Expanding the Experimental Setup
We elaborate how the experiments could be further expanded. There are three mayor ways to reach them. Based on further interviews, we could include more properties in the specification. Especially for the high-level reasoning, further properties are specifiable, e.g. collision detection for polygons. Furthermore, approximations for more involved properties such as pattern recognition, which are currently handled by neural networks, could be examined. Longer and computationally more intensive missions could be carried out and analyzed. Further, combinations of these points are possible.

Improving Unobstrusivness
In general, threads are suboptimal for the unobtrusiveness of the system since they are directly related to the system under scrutiny. Outsourcing the monitoring into an external process is the simplest way to cope with this problem. The process could be run on the same PC or on an separate (monitor-only) PC. Preferably, the external process receives the system values via a software or a hardware bus. Hardware monitors could further increase unobtrusiveness.

Monitoring of Asynchronous Sensors
We depicted the contingency manager in Section 3.2. In order to identify error sources, it can be useful to compare sensors arguing about similar system or environment parts. In order to do so, Lola has to support the merging of different sensor values which may be received asynchronously. In offline analysis, merging can be based on timestamps. In online analysis, it might be beneficial to further invest the method `synchronize_values` of the monitor interface. This method

allows to transform several values of a stream into a single one based on a predefined function. So far, we only considered the most recent value but considering the maximum or average over multiple values should be further investigated.

Language Extensions
Considering extensions to the LOLA specification language, we already mentioned sets and lists. Lately, a parametric LOLA version was published [25]. There, based on invoked template stream instances, data can be carried along the stream. For instance, we can monitor that *arbitrarily* many opened files, each with a unique ID, are all closed eventually. For each opened file, a template instance is invoked with the ID as key to identify its closing. In future, real time reasoning with LOLA would be an interesting language feature. Real time bounds in the offset operator, e.g. `trigger s[0..10sec, true]` could specify that within ten seconds *true* should hold for the stream `s`.

Active Usage
Motivating engineers to use LOLA in practice is not an easy task. Many engineers are included in several projects and their time to try out new tools is limited. Presenting the analysis power and error findings might be a way to encourage engineer to consider LOLA for their projects. Further, implementing the usage of the dependency graph and macros (LOLA library) could also be helpful to ease the writing of LOLA specifications. Using the dependency graph, a multithreaded version of LOLA based on the strongly connected components could potentially increase the performance. Additional, python plotting tools for online visualization could be useful to illustrate the usage.

8 Appendix

A Complete Extended Lola Syntax

⟨*lola-format*⟩ ::= ϵ | ⟨*streamDef*⟩ ⟨*lola-format*⟩

⟨*streamDef*⟩ ::= ⟨*inputDef*⟩
| ⟨*constantDef*⟩
| ⟨*outputDef*⟩
| ⟨*observableBehavior*⟩
| ⟨*knowledge*⟩
| ⟨*macroDef*⟩

⟨*constantDef*⟩ ::= '`const`' ⟨*type*⟩ ⟨*identifier*⟩ '`:=`' ⟨*literal*⟩

⟨*inputDef*⟩ ::= '`input`' ⟨*basicType*⟩ ⟨*identifier_list*⟩

⟨*outputDef*⟩ ::= '`output`' ⟨*type*⟩ ⟨*identifier*⟩ '`:=`' ⟨*expr*⟩

⟨*identifier_list*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩ '`,`' ⟨*identifier_list*⟩

Syntax 13: Structure

⟨*type*⟩ ::= ⟨*basicType*⟩ | ⟨*tuple*⟩

⟨*basicType*⟩ ::= ‘`bool`’ | ‘`int`’ | ‘`double`’ | ‘`string`’

⟨*tuple*⟩ ::= ‘`(`’ ⟨*tupleType*⟩ ‘`)`’

⟨*tupleType*⟩ ::= ⟨*basicType*⟩ **[** ‘`,`’ ⟨*tupleType*⟩ **]**
| ⟨*tuple*⟩ **[** ‘`,`’ ⟨*tupleType*⟩ **]**

⟨*identifier*⟩ ::= **(** ‘`a`’-‘`z`’|‘`A`’-‘`Z`’ **)** **(**‘`a`’-‘`z`’|‘`A`’-‘`Z`’ | ‘`0`’-‘`9`’ | ‘`_`’**)***

⟨*literal*⟩ ::= ⟨*boolLiteral*⟩ | ⟨*intLiteral*⟩ | ⟨*doubleLiteral*⟩
| ⟨*stringLiteral*⟩ | ⟨*tupleLiteral*⟩

⟨*boolLiteral*⟩ ::= ‘`true`’ | ‘`t`’ | ‘`false`’ | ‘`f`’

⟨*intLiteral*⟩ ::= **[** `-` **]** **(** ‘`0`’-‘`9`’ **)** **(** ‘`0`’-‘`9`’ **)***

⟨*doubleLiteral*⟩ ::= **[** `-` **]** **(** ‘`0`’-‘`9`’ **)** **(** ‘`0`’-‘`9`’ **)*** **[**`.`**]** **(** ‘`0`’-‘`9`’ **)***

⟨*stringLiteral*⟩ ::= any string starting with ‘`"`’ and ending with ‘`"`’

⟨*tupleLiteral*⟩ ::= ‘`(`’ ⟨*tupleValue*⟩ ‘`)`’

⟨*tupleValue*⟩ ::= ⟨*literal*⟩ | ⟨*literal*⟩ ‘`,`’ ⟨*tupleValue*⟩

Syntax 14: Literals

⟨*expr*⟩ ::= ⟨*literal*⟩ | '(' ⟨*expr*⟩ ')' | ⟨*unaryOp*⟩
| ⟨*binaryOp*⟩ | ⟨*ifExpr*⟩ | ⟨*switchExp*⟩
| ⟨*keyword*⟩ | ⟨*streamAccess*⟩ | ⟨*functionOp*⟩

⟨*keyword*⟩ ::= 'position' | 'last_position'
| 'int_min' | 'int_max'
| 'double_min' | 'double_max'

⟨*expressions*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ',' ⟨*expressions*⟩

Syntax 15: Expression

⟨*unaryOp*⟩ ::= '!' ⟨*expr*⟩

⟨*binaryOp*⟩ ::= ⟨*expr*⟩ ⟨*comparison*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*computation*⟩ ⟨*expr*⟩

⟨*comparison*⟩ ::= '<' | '<=' | '=' | '!=' | '>=' | '>'

⟨*computation*⟩ ::= '&' | '|' | '->' | '<-' | '<->'
| '+' | '-' | '*' | '/' | '%' | '^'

Syntax 16: Operators

left-associative	right-associative
	^
*, \, %	
+, −	
&	
|	
←, →	
↔	
<, ≤, =, ≠, ≥, >	

Figure 14: Operator Precedence, highest at the top.

⟨*functionOp*⟩ ::= ⟨*numberFunction*⟩ | ⟨*stringFunction*⟩ | ⟨*tupleOp*⟩

⟨*numberFunction*⟩ ::= ‘`abs`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`difference`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`atan2`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`max`’ ‘`(`’ ⟨*expressions*⟩ ‘`)`’ | ‘`min`’ ‘`(`’ ⟨*expressions*⟩ ‘`)`’
| ‘`sqrt`’ ‘`(`’ ⟨*expr*⟩ **[**‘`,`’ ⟨*doubleLiteral*⟩**]**‘`)`’
| ‘`log`’ ‘`(`’ ⟨*expr*⟩ **[**‘`,`’ ⟨*doubleLiteral*⟩**]** ‘`)`’
| ‘`cos`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ | ‘`sin`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`tan`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ | ‘`bin_to_int`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`int`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ | ‘`double`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`ceil`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ | ‘`floor`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`round`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’

⟨*stringFunction*⟩ ::= ‘`contains`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`equals`’ ‘`(`’ ⟨*expressions*⟩ ‘`)`’
| ‘`startswith`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`endswith`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`concat`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`length`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’

⟨*tupleOp*⟩ ::= ‘`get`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*intLiteral*⟩ ‘`)`’
| ‘`extract`’‘`(`’⟨*expr*⟩‘`,`’⟨*intLiteral*⟩**[**‘`,`’⟨*intLiteral*⟩**]***‘`)`’
| ‘`combine`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’

Syntax 17: Functions

⟨*ifExpr*⟩ ::= 'if' ⟨*expr*⟩ '{' ⟨*expr*⟩ '}' ⟨*elseExpr*⟩

⟨*elseExpr*⟩ ::= ['elif' ⟨*expr*⟩ '{' ⟨*expr*⟩ '}' ⟨*elseExpr*⟩]
'else' '{' ⟨*expr*⟩ '}'

⟨*switchExpr*⟩ ::= 'switch' ⟨*expr*⟩ '{' ⟨*cases*⟩ ⟨*default*⟩ '}'

⟨*cases*⟩ ::= ϵ
| 'case' ⟨*literal*⟩ '{' ⟨*expr*⟩ '}'
| 'case' ⟨*literal*⟩ '{' ⟨*expr*⟩ '}' ⟨*cases*⟩

⟨*default*⟩ ::= 'default' '{' ⟨*expr*⟩ '}'

Syntax 18: Statements

⟨*streamAccess*⟩ ::= ⟨*identifier*⟩ [⟨*relativePos*⟩ | ⟨*absolutePos*⟩]

⟨*relativePos*⟩ ::= ⟨*offset*⟩ | ⟨*frozenOffset*⟩
| ⟨*window*⟩ | ⟨*frozenWindow*⟩

⟨*absolutePos*⟩ := '#' ⟨*offset*⟩ | '#' ⟨*window*⟩

Syntax 19: Streamaccess Operators

⟨*offset*⟩ ::= '[' ⟨*offsetValues*⟩ ',' ⟨*oobV*⟩ ']'

⟨*frozenOffset*⟩ ::= '[' ⟨*offsetValues*⟩ ',' ⟨*oobV*⟩ ',' ⟨*frozenTime*⟩ ']'

⟨*window*⟩ ::= '[' ⟨*offsetValues*⟩ '..' ⟨*offsetValues*⟩
',' ⟨*oobV*⟩ ',' ⟨*binaryOp*⟩ ']'

⟨*frozenWindow*⟩ ::= '[' ⟨*offsetValues*⟩ '..' ⟨*offsetValues*⟩
',' ⟨*oobV*⟩ ',' ⟨*binaryOp*⟩ ',' ⟨*frozenTime*⟩ ']'

⟨*oobV*⟩ ::= ⟨*identifier*⟩ | ⟨*literal*⟩

⟨*frozenTime*⟩ ::= ⟨*identifier*⟩ | ⟨*literal*⟩

⟨*offsetValues*⟩ ::= (⟨*intLiteral*⟩ | 'last_position')

Syntax 20: Offset Operators

⟨*observableBehavior*⟩ ::= ⟨*onlineBehavior*⟩ ['with' ⟨*stringLiteral*⟩]
| ⟨*offlineBehavior*⟩ 'at' ⟨*location*⟩
| ⟨*controlCommands*⟩ ['with' ⟨*stringLiteral*⟩]

⟨*onlineBehavior*⟩ ::= 'trigger' ⟨*condition*⟩
| 'trigger_once' ⟨*condition*⟩
| 'trigger_change' ⟨*condition*⟩
| 'print' ⟨*identifier*⟩
| 'snapshot' ⟨*condition*⟩

⟨*offlineBehavior*⟩ ::= 'filter' ⟨*identifier_list*⟩ 'if' ⟨*condition*⟩
| 'tag' 'as' ⟨*identifier_list*⟩ 'if' ⟨*condition*⟩
'with' ⟨*identifier_list*⟩

⟨*controlCommands*⟩ ::= 'exit' ⟨*condition*⟩
| 'reset' ⟨*condition*⟩
| 'pause' ⟨*condition*⟩

⟨*condition*⟩ ::= ⟨*identifier*⟩ [⟨*comparison*⟩ ⟨*literal*⟩]

⟨*location*⟩ ::= ⟨*stringLiteral*⟩

Syntax 21: Observable Behavior

⟨*priorKnowledge*⟩ ::= ⟨*streamKnowledge*⟩ | ⟨*specificationKnowledge*⟩

⟨*streamKnowledge*⟩ ::= '`monotone_inc`' ⟨*identifier_list*⟩
| '`monotone_dec`' ⟨*identifier_list*⟩
| '`past_only`' ⟨*identifier_list*⟩
| '`future_only`' ⟨*identifier_list*⟩

⟨*specificationKnowledge*⟩ ::= '`efficient_fragment`'
| '`inefficient_fragment`'
| '`evaluation_order`' ⟨*identifier_list*⟩

Syntax 22: Prior Knowledge

⟨*macroDef*⟩ ::= '`macro`' ⟨*identifier*⟩ '`(`' ⟨*typed_parameterList*⟩'`)`'
'`{`' ⟨*macroBody*⟩ '`}`'

⟨*typed_parameterList*⟩ ::= ϵ
| '`stream`' ⟨*type*⟩ ⟨*identifier*⟩
| '`expression`' ⟨*expr*⟩
| ⟨*type*⟩ ⟨*identifier*⟩
| ⟨*type*⟩ ⟨*identifier*⟩ '`,`' ⟨*typed_parameterList*⟩

⟨*macroBody*⟩ ::= ϵ | ⟨*macroBody*⟩ ⟨*lola-format*⟩

Syntax 23: Macros

References

[1] Amazon Prime Air. https://www.amazon.com/b?node=8037720011. Online accessed: 2016-08-19.

[2] dSPACE. https://www.dspace.com. Online accessed: 2016-10-31.

[3] German Aerospace Center. http://www.dlr.de. Online accessed: 2016-08-19.

[4] Haversine formula. http://www.movable-type.co.uk/scripts/latlong.html. Online accessed: 2016-10-28.

[5] internet.org by facebook. https://info.internet.org/en/. Online accessed: 2016-08-19.

[6] Mantis bugtracker. http://www.mantisbt.org. Online accessed: 2016-08-19.

[7] Nasa. https://www.nasa.gov. Online accessed: 2016-10-19.

[8] Project skybender. http://shortlinks.de/wl4i. Online accessed: 2016-08-19.

[9] Scala. http://www.scala-lang.org. Online accessed: 2016-10-18.

[10] Subversion. https://subversion.apache.org. Online accessed: 2016-08-19.

[11] time command. http://man7.org/linux/man-pages/man1/time.1.html. Online accessed: 2016-10-31.

[12] Florian M. Adolf and Frank Thielecke. A sequence control system for onboard mission management of an unmanned helicopter. *American Institute of Aeronautics and Astronautics, Inc.*, 2007.

[13] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 15–40, 2016.

[14] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993.

[15] Howard Barringer, Klaus Havelund, David E. Rydeheard, and Alex Groce. Rule systems for runtime verification: A short tutorial. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, pages 1–24, 2009.

[16] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, pages 260–272, 2006.

[17] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, pages 126–138, 2007.

[18] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 43(1):29–60, 2013.

[19] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 353–362, 1989.

[20] Konstantinos Dalamagkidis, Kimon P. Valavanis, and Les A. Piegl. On integrating unmanned aircraft systems into the national airspace system: Issues, challenges, operational restrictions, certification, and ... and automation science and engineering). *Springer Netherlands, Dordrecht, 2012. ISBN 978-94-007-2478-5. doi: 10.1007/978-94-007-2479-2*, 2012.

[21] Werner Damm and Bernd Finkbeiner. Automatic compositional synthesis of distributed systems. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 179–193, 2014.

[22] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174, 2005.

[23] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[24] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, pages 141–175. 2013.

[25] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2016.

[26] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27(3):253–274, November 2005.

[27] Deutsche Flugsicherung. Nachrichten für Luftfahrer Gemeinsame Grundsätze des Bundes und der Länder für die Erteilung der Erlaubnis zum Aufstieg von unbemannten Luftfahrtsystemen gemäß Paragraph 16 Absatz 1 Nummer 7 Luftverkehrs-Ordnung (LuftVO). 60. Jahrgang, Langen, NfL I 161. 2012.

[28] Johannes Geist, Kristin Y. Rozier, and Johann Schumann. Runtime observer pairs and bayesian network reasoners on-board fpgas: Flight-certifiable system health management for embedded systems. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 215–230, 2014.

[29] Sylvain Hallé. When RV meets CEP. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 68–91, 2016.

[30] Mark Timon Hüneberg. Optimizing LOLA specifications. Bachelor Thesis 2015.

[31] Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Rosu. Efficient monitoring of parametric context-free patterns. *Autom. Softw. Eng.*, 17(2):149–180, 2010.

[32] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for C programs. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 603–606, 2013.

[33] O.Hirling and F.Holzapfel. Applicability of military uas airworthiness regulations to civil fixed wing light uas in germany. *Deutsche Gesellschaft fÃijr Luft- und Raumfahrt - Lilienthal-Oberth e.V., Bonn, 2012*, 2012.

[34] The Joint JAA/EUROCONTROL Initiative on UAVs. Uav task-force final report a concept for european regulations for civil unmanned aerial vehicles (uavs). 2004.

[35] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[36] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 357–372, 2014.

[37] RTCA. Do-178c/ed-12c software considerations in airborne systems and equipment certification. 2011.

[38] RTCA. Do-331/ed-218 model-based development and verification supplement to do-178c and do-278a. 2011.

[39] Johann Schumann, Timmy Mbaya, Ole J. Mengshoel, Knot Pipatsrisawat, Ashok N. Srivastava, Arthur Choi, and Adnan Darwiche. Software health management with bayesian networks. *ISSE*, 9(4):271–292, 2013.

[40] Johann Schumann, Kristin Y. Rozier, Thomas Reinbacher, Ole J. Mengshoel, Timmy Mbaya, and Corey Ippolito. Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *Annual Conference of the Prognostics and Health Management Society*, 2013.

[41] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.

[42] Christoph Torens, Florian Adolf, Girish Patil, and Ganesh K. Vernekar. Towards generic requirements and models for automated mission tasks with rpas. *American Institute of Aeronautics and Astronautics, Inc.*, 2016.

[43] Christoph Torens and Florian M. Adolf. Automated verification and validation of an onboard mission planning and execution system for UAVs. *American Institute of Aeronautics and Astronautics, Inc.*, 2013.

[44] Christoph Torens and Florian M. Adolf. Software verification considerations for the ARTIS unmanned rotorcraft. *In book: 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, 2013.

[45] Christoph Torens and Florian M. Adolf. Formal requirements and model-checking for v & v automation of a RPAS mission management system. *American Institute of Aeronautics and Astronautics*, 2015.

[46] Christoph Torens, Florian M. Adolf, and Lukas Goormann. Certification and software verification considerations for autonomous unmanned aircraft. *Journal of Aerospace Information Systems*, 2014.

[47] C. Watterson and Donal Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, 2007.

[48] W. Eric Wong, Joseph R. Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering, ISSRE 1997, Albuquerque, NM, USA, November 2-5, 1997*, pages 264–274, 1997.

[49] Chun Wah Wallace Wu, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. Reducing monitoring overhead by integrating event- and time-triggered techniques. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 304–321, 2013.