RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft

Jan Baumeister[1 [0000−0002−8891−7483]], Bernd Finkbeiner[1 [0000−0002−4280−8441]], Sebastian Schirmer[2], Maximilian Schwenger[1 [0000−0002−2091−7575]], and Christoph Torens[2]

[1] Saarland University, Department of Computer Science, 66123 Saarbrücken, Germany
{jbaumeister, finkbeiner, schwenger}@react.uni-saarland.de
[2] German Aerospace Center (DLR), 38108 Braunschweig, Germany
{christoph.torens, sebastian.schirmer}@dlr.de

Abstract. The autonomous control of unmanned aircraft is a highly safety-critical domain with great economic potential in a wide range of application areas, including logistics, agriculture, civil engineering, and disaster recovery. We report on the development of a dynamic monitoring framework for the DLR ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) family of unmanned aircraft based on the formal specification language RTLola. RTLola is a stream-based specification language for real-time properties. An RTLola specification of hazardous situations and system failures is statically analyzed in terms of consistency and resource usage and then automatically translated into an FPGA-based monitor. Our approach leads to highly efficient, parallelized monitors with formal guarantees on the noninterference of the monitor with the normal operation of the autonomous system.

Keywords: Runtime Verification, Stream Monitoring, FPGA, Autonomous Aircraft

1 Introduction

An unmanned aerial vehicle, commonly known as a drone, is an aircraft without a human pilot on board. While usually connected via radio transmissions to a base station on the ground, such aircraft are increasingly equipped with decision-making capabilities that allow them to autonomously carry out complex missions in applications such as transport, mapping and surveillance, or crop and irrigation monitoring. Despite the obvious safety-criticality of such systems, it is impossible to foresee all situations an autonomous aircraft might encounter and thus make a safety case purely by analyzing all of the potential behaviors in advance. A critical part of the safety engineering of a drone is therefore to carefully monitor the actual behavior during the flight, so that the health status of the system can be assessed and mitigation procedures (such as a return to the base station or an emergency landing) can be initiated when needed.

In this paper, we report on the development of a dynamic monitoring framework for the DLR ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) family of aircraft based on the formal specification language RTLola. The development of a monitoring framework for an autonomous aircraft differs significantly from a monitoring framework in a more standard setting, such as network monitoring. A key consideration is that while the specification language needs to be highly *expressive*, the monitor must operate within strictly limited resources, and the monitor itself needs to be highly *reliable*: any interference with the normal operation of the aircraft could have fatal consequences.

A high level of expressiveness is necessary because the assessment of the health status requires complex analyses, including a cross-validation of different sensor modules such as the agreement between the GPS module and the accelerometer. This is necessary in order to discover a deterioration of a sensor module. At the same time, the expressiveness and the precision of the monitor must be balanced against the available computing resources. The reliability requirement goes beyond pure correctness and robustness of the execution. Most importantly, reliability requires that the peak resource consumption of the monitor in terms of energy, time, and space needs to be known ahead of time. This means that it must be possible to compute these resource requirements statically based on an analysis of the specification. The determination whether the drone is equipped with sufficient hardware can then be made before the flight, and the occurrence of dynamic failures such as running out of memory or sudden drops in voltage can be ruled out. Finally, the collection of the data from the on-board architecture is a non-trivial problem: While the monitor needs access to almost the complete system state, the data needs to be retrieved non-intrusively such that it does not interfere with the normal system operation.

Our monitoring approach is based on the formal stream specification language RTLola [11]. In an RTLola specification, input streams that collect data from sensors, networks, etc., are filtered and combined into output streams that contain data aggregated from multiple sources and over multiple points in time such as over sliding windows of some real-time length. Trigger conditions over these output streams then identify critical situations. An RTLola specification is translated into a monitor defined in a hardware description language and subsequently realized on an FPGA. Before deployment, the specification is checked for consistency and the minimal requirements on the FPGA are computed. The hardware monitor is then placed in a central position where as much sensor data as possible can be collected; during the execution, it then extracts the relevant information. In addition to requiring no physical changes to the system architecture, this integration incurs no further traffic on the bus.

Our experience has been extremely positive: Our approach leads to highly efficient, parallelized monitors with formal guarantees on the non-interference of the monitor with the normal operation of the autonomous system. The monitor is able to detect violations to complex specifications without intruding into the system execution, and operates within narrow resource constraints. RTLola is cleared for take-off.

1.1 Related Work

Stream-based monitoring approaches focus on an expressive specification language while handling non-binary data. Its roots lie in synchronous, declarative stream processing languages like Lustre [13] and Lola [8]. The *Copilot* framework [19] features a declarative data-flow language from which constant space and constant time C monitors are generated; these guarantees enable usage on an embedded device. Rather than focusing on data-flow, the family of Lola-languages puts an emphasis on statistical measures and has successfully been used to monitor synchronous, discrete time properties of autonomous aircraft [1,23]. In contrast to that, RTLola [12,22] supports real-time capabilities and efficient aggregation of data occurring with arbitrary frequency, while forgoing parametrization for efficiency [11]. RTLola can also be compiled to VHDL and subsequently realized on an FPGA [7].

Apart from stream-based monitoring, there is a rich body of monitoring based on real-time temporal logics [2,9,14–16,20] such as Signal Temporal Logic (STL) [17]. Such languages are a concise way to describe temporal behaviors with the shortcoming that they are usually limited to qualitative statements, i.e. boolean verdicts. This limitation was addressed for STL [10] by introducing a quantitative semantics indicating the robustness of a satisfaction. To specify continuous signal patterns, specification languages based on regular expressions can be beneficial, e.g. Signal Regular Expressions (SRE) [5]. The R2U2 tool [18] stands out in particular as it successfully brought a logic closely related to STL onto unmanned aerial systems as an external hardware implementation.

2 Setup

The Autonomous Rotorcraft Testbed for Intelligent Systems (ARTIS) is a platform used by the Institute of Flight Systems of the German Aerospace Center (DLR) to conduct research on autonomous flight. It consists of a set of unmanned helicopters and fixed-wing aircraft of different sizes which can be used to develop new techniques and evaluate them under real-world conditions.

The case study presented in this paper revolves around the superARTIS, a large helicopter with a maximum payload of 85kg, depicted in Fig. 1. The high payload capabilities allow the aircraft to carry multiple sensor systems, computational resources, and data links. This extensive range of avionic equipment plays an important role in improving the situational awareness of the aircraft [3] during the flight. It facilitates safe autonomous research missions which include flying in urban or maritime areas, alone or with other aircraft. Before an actual flight test, software- and hardware-in-the-loop simulations, as well as real-time logfile replays strengthen confidence in the developed technology.

2.1 Mission

One field of application for unmanned aerial vehicles (UAVs) is reconnaissance missions. In such missions, the aircraft is expected to operate within a fixed area

in which it can cause no harm. The polygonal boundary of this area is called a geo-fence. As soon as the vehicle passes the geo-fence, mitigation procedures need to be initiated to ensure that the aircraft does not stray further away from the safe area.

The case study presented in this paper features a reconnaissance mission. Fig. 2 shows the flight path (blue line) within a geo-fence (red line). Evidently, the aircraft violates the fence several times temporarily. A reason for this can be flawed position estimation: An aircraft estimates its position based on several factors such as landmarks detected optically or GPS sensor readings. In the latter case, GPS satellites send position and time information to earth. The GPS module uses this data to compute the aircraft's absolute position with trilateration. However, signal reflection or a low number of GPS satellites in range can result in imprecisions in the position approximation. If the aircraft is continuously exposed to imprecise position updates, the error adds up and results in a strong deviation from the expected flight path.

The impact of this effect can be seen in Fig. 3. It shows the velocity of a ground-borne aircraft in an enclosed backyard according to its GPS module.[3] During the reported period of time, the aircraft was pushed across the backyard by hand. While the expected graph is a smooth curve, the actual measurements show an erratic curve with errors of up to $\pm 1.5 \mathrm{ms}^{-1}$, which can be mainly attributed to signals being reflected on the enclosure. The strictly positive trend of the horizontal velocity can explain strong deviations from the desired flight path seen in Fig. 3.

A counter-measure to these imprecisions is the cross-validation of several redundant sensors. As an example, rather than just relying on the velocity reported by a GPS module, its measured velocity can be compared to the integrated output of an accelerometer. When the values deviate strongly, the values can be classified as less reliable than when both sensors agree.

2.2 Non-Intrusive Instrumentation

When integrating the monitor into an existing system, the system architecture usually cannot be altered drastically. Moreover, the monitor should not interfere with the regular execution of the system, e.g. by requiring the controller to send explicit messages to it. Such a requirement could offset the timing behavior and thus have a negative impact on the overall performance of the system.

The issue can be circumvented by placing the monitor at a point where it can access all data necessary for the monitoring process non-intrusively. In the case of the superARTIS, the logger interface provides such a place as it compiled the data of all position-related sensors as well as the output of the position estimation [3, 4]. Fig. 4 outlines the relevant data lines of the aircraft. Sensors were polled with fixed frequencies of up to 100Hz. The schematic shows that the logger explicitly sends data to the monitor. This is not a strict requirement of

[3] GPS modules only provide absolute position information; the first derivative thereof, however, is the velocity.

Fig. 1: DLR's autonomous superARTIS equipped with optical navigation.

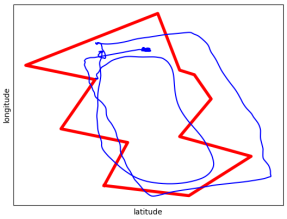


Fig. 2: Reconnaissance mission for a UAV. The thin blue line represents its trajectory, the thick red line a geofence.

the monitor as it could be connected to the data busses leading to the logger and passively read incoming data packets. However, in the present setting, the logger did not run at full capacity. Thus sending information to the monitor came at no relevant cost while requiring few hardware changes to the bus layout.

In turn, the monitor provides feedback regarding violations of the specification. Here, we distinguish between different timing behaviors of triggers. The monitor evaluates event-based triggers whenever the system passes new events to the monitor and immediately replies with the results. For periodic triggers, i.e., those annotated with an evaluation frequency, the evaluation is decoupled from the communication between monitor and system. Thus, the monitor needs to wait until it receives another event until reporting the verdict. This incurs a short delay between detection and report.

2.3 StreamLAB

StreamLAB[4] [11] is a monitoring framework revolving around the stream-based specification language RTLola. It emphasizes on analyses conducted before deployment of the monitor. This increases the confidence in a successful execution by providing information to aid the specifier. To this end, it detects inconsistencies in the specification such as type errors, e.g. an lossy conversion of a floating point number to an integer, or timing errors, e.g. accessing values that might not exist. Further, it provides two execution modes: an interpreter and an FPGA compilation. The interpreter allows the specifier to validate their specification. For this, it requires a *trace*, i.e. a series of data that is expected to occur during an execution of the system. It then checks whether a trace complies with the specification and reports the points in time when specified bounds are violated. After successfully validating the specification, it can be compiled into

[4] stream-lab.eu

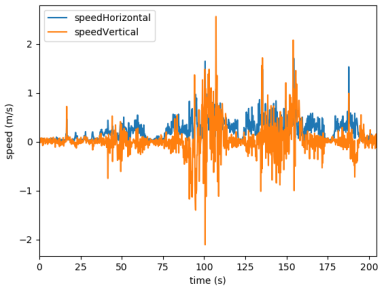


Fig. 3: Line plot of the horizontal and vertical speed calculated by a GPS receiver.

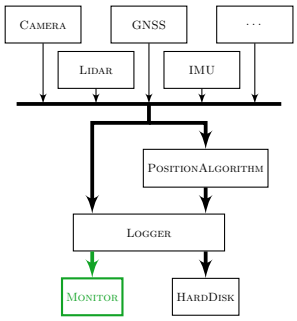


Fig. 4: Overview of data flow in system architecture.

VHDL code. Yet again, the compiled code can be analyzed with respect to the space and power consumption. This information allows for evaluating whether the available hardware suffices for running the RTLola monitor.

An RTLola specification consists of input and output streams, as well as trigger conditions. *Input* streams describe data the system produces asynchronously and provides to the monitor. *Output* streams use this data to assess the health state of the system e.g. by computing statistical information. *Trigger* conditions distinguish desired and undesired behavior. A violation of the condition issues an alarm to the system.

The following specification declares a floating point input stream `height` representing sensor readings of an altimeter. The output stream `avg_height` computes the average value of the `height` stream over two minutes. The aggregation is a sliding window computed once per second, as indicated with the `@1Hz` annotation.[5] The stream `δheight` computes the difference between the average and the current height. A strong deviation of these values constitutes a suspicious jump in sensor readings, which might indicate a faulty sensor or an unexpected loss or gain in height. In this case, the trigger in the specification issues a warning to the system, which can initiate mitigation measures.

```
input height: Float32
output avg_height @1Hz := height.aggregate(over: 2min, using: avg)
output δheight := abs(avg_height.hold().defaults(to: height) - height)
trigger δheight > 50.0 "WARNING: Suspicious jump in height."
```

Note that this is just a brief introduction to RTLola and the StreamLAB framework. For more details, the authors refer to [7, 11, 12, 22].

[5] Details on how such a computation can cope with a statically-bounded amount of memory can be found in [12, 22].

2.4 FPGA as Monitoring Platform

An RTLola specification can be compiled into the hardware description language VHDL and subsequently realized on an FPGA as proposed by Baumeister et al. [7]. An FPGA as target platform for the monitor has several advantages in terms of improving the development process, reducing its cost, and increasing the overall confidence in the execution.

Since the FPGA is a separate module and thus decoupled from the control software, these components do not share processor time or memory. This especially means that control and monitoring computations happen in parallel. Further, the monitor itself parallelizes the computation of independent RTLola output streams with almost no additional overhead. This significantly accelerates the monitoring process [7]. The compiled VHDL specification allows for extensive static analyses. Most notably, the results include whether the board is sufficiently large in terms of look-up tables and storage capabilities to host the monitor, and the power consumption when idle or at peak performance. Lastly, an FPGA is the sweet spot between generality and specificity: it runs faster, is lighter, and consumes less energy than general purpose hardware while retaining a similar time-to-deployment. The latter combined with a drastically lower cost renders the FPGA superior to application-specific integrated circuits (ASIC) during development phase. After that, when the specification is fixed, an ASIC might be considered for its yet increased performance.

2.5 RTLola Specifications

The entire specification for the mission is comprised of three sub-specifications. This section briefly outlines each of them and explains representative properties in Fig. 5. The complete specifications as well as a detailed description were presented in earlier work [6, 21].

Sensor Validation (Appendix A.1) Sensors can produce incorrect values, e.g. when too few GPS satellites are in range for an accurate trilateration or if the aircraft flies above the range of a radio altimeter. A simple exemplary validation is to check whether the measured altitude is non-negative. If such a check fails, the values are meaningless, so the system should not take them into account in its computations.

Geo-Fence (Appendix A.2) During the mission, the aircraft has permission to fly inside a zone delimited by a polygon, called a geo-fence. The specification checks whether a face of the fence has been crossed, in which case the aircraft needs to ensure that it does not stray further from the permitted zone.

Sensor Cross-Validation (Appendix A.3) Sensor redundancy allows for validating a sensor reading by comparing it against readings of other sensors. An agreement between the values raises the confidence in their correctness. An example is the cross-validation of the GPS module against the accelerometer. Integrating the readings of the latter twice yields an absolute position which can be compared against the GPS position.

```
input gps_x: Float16 // Absolute x positive from GPS module
input num_sat : UInt8 // Number of GPS satellites in range
input imu_acc_x: Float32 // Acceleration in x direction from IMU
// Check if the GPS module emitted few readings in the last 3s.
trigger @1Hz gps_x.aggregate(over: 3s, using: count) < 10
  "VIOLATION: Few GPS updates "
// 1 if there are few GPS Satellites in range, otherwise 0.
output few_sat: UInt8 := Int(num_sat < 9)
// Check if there rarely were enough GPS satellites in range.
trigger @1Hz few_sat.aggregate(over: 5s, using: Σ) > 12 "WARNING:
    Unreliable GPS data."
// Integrate acceleration twice to obtain absolute position.
output imu_vel_x@1Hz := imu_acc_x.aggregate(over: ∞, using: ∫)
output imu_x@1Hz := imu_vel_x.aggregate(over: ∞, using: ∫)
// Issue an alarm if readings from GPS and IMU disagree.
trigger abs(imu_x − gps_x) > 0.5 "VIOLATION: GPS and IMU readings
    deviate."
```

Fig. 5: An RTLola specification validating GPS sensor data and cross validating readings from the GPS module and IMU.

Fig. 5 points out some representative sub-properties of the previously described specification in RTLola, which are too long to discuss them in detail. It contains a validation of GPS readings as well as a cross-validation of the GPS module against the Inertial Measurement Unit (IMU). The specification declares three input streams, the x-position and number of GPS satellites in range from the GPS module, and the acceleration in x-direction according to the IMU.

The first trigger counts the number of updates received from the GPS module by counting how often the input stream `gps_x` gets updated to validate the timing behavior of the module.

The output stream `few_sat` computes the indicator function for `num_sat < 9`, which indicates that the GPS module might report unreliable data due to few satellites in reach. If this happens more than 12 times within five seconds, the next trigger issues a warning to indicate that the incoming GPS values might be inaccurate. The last trigger checks whether the double integral of the IMU acceleration coincides with the GPS position up to a threshold of 0.5 meters.

2.6 VHDL Synthesis

The specifications mentioned above were compiled into VHDL and realized on the Xilinx ZC702 Base Board[6]. The following table details the resource consumption of each sub-specification reported by the synthesis tool Vivado.

[6] https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

Spec	FF	FF[%]	LUT	LUT[%]	MUX	Idle [mW]	Peak [W]
Geo-fence	2,853	3	26,181	71	4	149	1.871
Validation	4,792	5	34,630	67	104	156	2.085
Cross	3,441	4	23,261	46	99	150	1.911

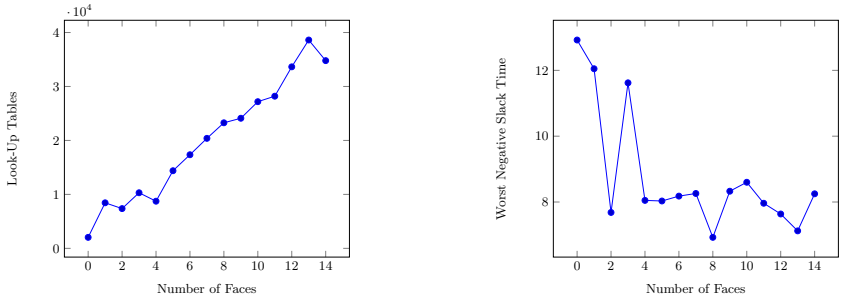
The number of flip-flops (FF) indicates the memory consumption in bits; neither specification requires more than 600B of memory. The number of LUTs (Look-up Tables) is an indicator for the complexity of the logic. The sensor validation, despite being significantly longer than the cross-validation, requires the least amount of LUTs. The reason is that its computations are simple in comparison: Rather than computing sliding window aggregations or line intersections, it mainly consists of simple thresholding. The number of multiplexers (MUX) reflects this as well: Since thresholding requires comparisons, which translate to multiplexers, the validation requires twice as many of them. Lastly, the power consumption of the monitor is extremely low: When idle, neither specification requires more than 156mW and even under peak pressure, the power consumption does not exceed 2.1W. For comparison, a Raspberry Pi needs between 1.1W (Model 2B) and 2.7W (Model 4B) when idle and roughly twice as much under peak pressure, i.e., 2.1W and 6.4W, respectively.[7]

Note that the geo-fence specification checks for 12 intersections in parallel, one for each face of the fence (cf. Fig. 2). Adapting the number of faces allows for scaling the amount of FPGA resources required, as can be seen in Fig. 6a. The graph does not grow linearly because the realization problem of VHDL code onto an FPGA is a multi-dimensional optimization problem with several pareto-optimal solutions. Under default settings, the optimizer found a solution for four faces that required fewer LUTs than for three faces. At the same time, the worst negative slack time (WNST) of the four-face solution was lower than the WNST for the three-face solution as well (cf. Fig. 6b), indicating that the former performs worst in terms of running time.

3 Results

As the title of the paper suggests, the superARTIS with the RTLola monitor component is cleared to fly and a flight test is already scheduled. In the meantime, the monitor was validated on log files from past missions of the superARTIS replayed under realistic conditions. During a flight, the controller polls samples from sensors, estimates the current position, and sends the respective data to the logger and monitor. In the replay setting, the process remains the same except for one detail: Rather than receiving data from the actual sensors, the data sent to the controller is read from a past log file in the same frequency in which they were recorded. The timing and logging behavior is equivalent to a real execution. This especially means that the replayed data points will be recorded again in

[7] Information collected from https://www.pidramble.com/wiki/benchmarks/power-consumption in January, 2020.



(a) Look-Up Tabels (b) Worst Negative Slack Time

Fig. 6: Result of the static analysis for different amounts of face of the geo-fence.

the same way. Control computations take place on a machine identical to the one on the actual aircraft. As a result, from the point of view of the monitor, the replay mode and the actual flight are indistinguishable. Note that the setup is open-loop, i.e., the monitor cannot influence the running system. Therefore, the replay mode using real data is more realistic than a high-fidelity simulation.

When monitoring the geo-fence of the reconnaissance mission in Fig. 2, all twelve face crossings were detected successfully. Additionally, when replaying the sensor data of the experiment in the enclosed backyard from Section 2.1, the erratic GPS sensor data lead to 113 violations regarding the GPS module on its own. Note that many of these violations point to the same culprit: a low number of available GPS satellites, for example, correlates with the occurrence of peaks in the GPS velocity. Moreover, the cross validation issued another 36 alarms due to a divergence of IMU and GPS readings. Other checks, for example detecting a deterioration of the GPS module based on its output frequency, were not violated in either flight and thus not reported.

4 Conclusion

We have presented the integration of a hardware-based monitor into the superARTIS UAV. The distinguishing features of our approach are the high level of expressiveness of the RTLola specification language combined with the formal guarantees on the resource usage. The comprehensive tool framework facilitates the development of complex specifications, which can be validated on log data before they get translated into a hardware-based monitor. The automatic analysis of the specification derives the minimal requirements on the development board needed for safe operation. If they are met, the specification is realized on an FPGA and integrated into the superARTIS architecture. Our experience shows that the overall system works correctly and reliably, even without thorough system-level testing. This is due to the non-interfering instrumentation, the validated specification, and the formal guarantees on the absence of dynamic failures of the monitor.

References

1. Adolf, F., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. In: Lahiri, S.K., Reger, G. (eds.) Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10548, pp. 33–49. Springer (2017). https://doi.org/10.1007/978-3-319-67531-2_3, https://doi.org/10.1007/978-3-319-67531-2_3
2. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. In: [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 390–401 (Jun 1990). https://doi.org/10.1109/LICS.1990.113764
3. Ammann, N., Andert, F.: Visual navigation for autonomous, precise and safe landing on celestial bodies using unscented kalman filtering. In: 2017 IEEE Aerospace Conference. pp. 1–12 (March 2017). https://doi.org/10.1109/AERO.2017.7943933
4. Andert, F., Ammann, N., Krause, S., Lorenz, S., Bratanov, D., Mejías, L.: Optical-aided aircraft navigation using decoupled visual SLAM with range sensor augmentation. Journal of Intelligent and Robotic Systems **88**(2-4), 547–565 (2017). https://doi.org/10.1007/s10846-016-0457-6, https://doi.org/10.1007/s10846-016-0457-6
5. Bakhirkin, A., Ferrère, T., Maler, O., Ulus, D.: On the quantitative semantics of regular expressions over real-valued signals. In: Abate, A., Geeraerts, G. (eds.) Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10419, pp. 189–206. Springer (2017). https://doi.org/10.1007/978-3-319-65765-3_11, https://doi.org/10.1007/978-3-319-65765-3_11
6. Baumeister: Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring. Master thesis, Saarland University (2020)
7. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embedded Comput. Syst. **18**(5s), 88:1–88:24 (2019). https://doi.org/10.1145/3358220, https://doi.org/10.1145/3358220
8. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174. IEEE Computer Society Press (June 2005)
9. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: FORMATS 2010. pp. 92–106. FORMATS'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1885174.1885183
10. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6246, pp. 92–106. Springer (2010). https://doi.org/10.1007/978-3-642-15297-9_9, https://doi.org/10.1007/978-3-642-15297-9_9
11. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24, https://doi.org/10.1007/978-3-030-25540-4_24

12. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017), http://arxiv.org/abs/1711.03829
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proceedings of the IEEE. pp. 1305–1320 (1991)
14. Harel, E., Lichtenstein, O., Pnueli, A.: Explicit clock temporal logic. In: LICS 1990. pp. 402–413. IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113765, https://doi.org/10.1109/LICS.1990.113765
15. Jahanian, F., Mok, A.K.L.: Safety analysis of timing properties in real-time systems. IEEE Transactions on Software Engineering **SE-12**(9), 890–904 (Sept 1986). https://doi.org/10.1109/TSE.1986.6313045
16. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
17. Maler, O., Nickovic, D.: Monitoring properties of analog and mixed-signal circuits. STTT **15**(3), 247–268 (2013). https://doi.org/10.1007/s10009-012-0247-9, https://doi.org/10.1007/s10009-012-0247-9
18. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Formal Methods in System Design **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x, https://doi.org/10.1007/s10703-017-0275-x
19. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A Hard Real-Time Runtime Monitor, pp. 345–359. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9˙26, http://dx.doi.org/10.1007/978-3-642-16612-9_26
20. Raskin, J.F., Schobbens, P.Y.: Real-time logics: Fictitious clock as an abstraction of dense time, pp. 165–182. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0035387
21. Schirmer, S., Torens, C., Adolf, F.: Formal Monitoring of Risk-based Geofences. https://doi.org/10.2514/6.2018-1986, https://arc.aiaa.org/doi/abs/10.2514/6.2018-1986
22. Schwenger, M.: Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS. Master thesis, Saarland University (2019)
23. Torens, C., Adolf, F., Faymonville, P., Schirmer, S.: Towards intelligent system health management using runtime monitoring. In: AIAA Information Systems-AIAA Infotech @ Aerospace. American Institute of Aeronautics and Astronautics (AIAA) (jan 2017). https://doi.org/10.2514/6.2017-0419, https://doi.org/10.2514%2F6.2017-0419

A RTLola Specifications

This section gives more details on the RTLola specification.

A.1 Sensor Validation

In this subsection, we consider GPS receiver submodules, namely: GPS velocity and position. In the specification shown in Fig. 7, the outputs of both modules are declared as inputs to the monitor (lines 5–6 and 15–18). Basic checks whether the absolute values of the horizontal and vertical speed are within bounds are performed in lines 9 and 10, where triggers check if any bound is exceeded. Similarly, valid states of the `position_type` (lines 21) and bounds for `diff_age` and `solution_age` (lines 23–24) are easily checked. Lines 26–33 evaluate the performance of the position estimation based on the number of satellites in reach. In line 27, we count once per second, i.e. with 1Hz, the number of satellites over the last three seconds. We expect to see at least nine satellites within this frame of time. Next, we trigger a notification on each rising edge of `behavior_num_sats` (line 32-33). The stream `behavior_num_sats` is an aggregation over five seconds where `curr_num_sats_invalid` represents an auxiliary stream which valides the number of satellites. Once per second we check if within the last five seconds more than twelve events reported a number of satellites below 9. If this is the case, we warn the system because the GPS modules seems to deteriorate.

The specification in Fig. 8 complements the GPS validation. Here, we focus on the horizontal (lines 9-11), vertical speed (lines 13-15), and the length of the speed vector (lines 17-20). Each second, we aggregate the respective speed and take the average. Each time we receive a new speed event, we compare its value against the computed average value of the last ten seconds. If there is a strong deviation, we raise a violation.

The validation specification is the combination of Fig. 7 and Fig. 8.

A.2 Geo-fence

In this section, we depict the detection of a single line crossing of a geo-fence. The specification can be seen in Fig. 9 where italic variables represent constant points of the geo-fence (*lat*, *lon*) and their respective slopes (m) and y-intercepts (b). The basic idea is to compute the intersection of the geo-fence line and the vehicle line, i.e. the line given the last position and the current position of the vehicle. The vehicle line is computed in lines 12–21. The slope of the line and the y-intercept are calculated in lines 20 and 21, respectively. Given these parameters, the computation of the intersection point is basic geometry, i.e. $y = m \cdot x + t$. Finally, we check whether the intersection point lies on both the vehicle line and the geo-fence face (lines 34-41). The output `is_fnc` is true if the vehicle movement can be encoded as a valid function (line 32).

```
import math

// GPS Velocity Submodule
// Input Streams
input speed_h : Float16 // Horizontal Speed
input speed_v : Float16 // Veritical Speed

// Bound checks
trigger abs(speed_h) > 1.5 "VIOLATION: Horizontal speed exceeds threshold."
trigger abs(speed_v) > 2.0 "VIOLATION: Vertical speed exceeds threshold."


// GPS Position Submodule
// Input Streams
input diff_age : Float32 // Time since last correction solution
input solution_age : Float16 // Elapsed time for solution
input position_type : UInt8 // Internal state
input num_sats : UInt8 // Number of received satellites signals

// Internal state validation
trigger ¬(position_type = 34 ∨ position_type = 17) "VIOLATION: Invalid position_type state."
// Solution Validation
trigger diff_age > 135.0 "VIOLATION: Correction too old."
trigger solution_age > 0.15 "VIOLATION: Solution took too long."

// Received satellites signal check
trigger @1Hz num_sats.aggregate(over: 3s, using: count) < 10
       "DEGRADATION: With given frequency, more satellites expected."
output cur_num_sats_invalid : UInt8 := if num_sats < 9 then 1 else 0
output behavior_num_sats : Bool @ 1Hz :=
                        cur_num_sats_invalid.aggregate(over: 5s,
                            using: sum) > 12
trigger @1Hz !behavior_num_sats.offset(by:-1).defaults(to:false) ∧ behavior_num_sats
  "DEGRADATION: Bad gps performance due to recent decline in number satellites."
```

Fig. 7: Sensor validation

```
import math

// GPS Velocity Submodule
// Input Streams
input speed_h : Float16 // Horizontal Speed
input speed_v : Float16 // Veritical Speed

// Peak detection
output avg_speed_h @1Hz := speed_h.aggregate(over: 10s, using:
    avg).defaults(to:0.0)
output speed_h_diff := abs(speed_h -
    avg_speed_h.hold().defaults(to:speed_h))
trigger speed_h_diff > 0.4 "VIOLATION: Peak in horizontal speed."

output avg_speed_v @1Hz := speed_v.aggregate(over: 10s, using:
    avg).defaults(to:0.0)
output speed_v_diff := abs(speed_v -
    avg_speed_v.hold().defaults(to:speed_v))
trigger abs_speed_v_diff > 1.0 "VIOLATION: Peak in vertical speed."

output speed_all := sqrt(speed_h * speed_h + speed_v * speed_v)
output avg_speed_all @1Hz := speed_all.aggregate(over: 10s, using:
    avg).defaults(to:0.0)
output speed_all_diff := abs(speed_all -
    avg_speed_all.hold().defaults(to:speed_all))
trigger speed_all_diff > 1.0 "VIOLATION: Peak in speed vector."
```

Fig. 8: Peak detection

```
import math

// Declaring Inputs
input lat_in_degree :Float32
input lon_in_degree :Float32

// Transform Degree in Radian
output lat := lat_in_degree * 3.14159265359 / 180.0
output lon := lon_in_degree * 3.14159265359 / 180.0

// Vehicle Line
output lat_pre := lat.offset(by: -1).defaults(to: lat)
output delta_lat := lat - lat_pre

output lon_pre := lon.offset(by: -1).defaults(to: lon)
output delta_lon := lon - lon_pre

output is_fnc := abs(delta_lat) > ϵ

output m_v := if is_fnc then (delta_lon) / (delta_lat) else 0.0
output b_v := if is_fnc then lon - (m_v * lat) else 0.0

output min_lat_v:= if lat < lat_pre then lat else lat_pre
output max_lat_v := if lat > lat_pre then lat else lat_pre

output min_lon_v := if lon < lon_pre then lon else lon_pre
output max_lon_v := if lon > lon_pre then lon else lon_pre

// Polygonline p1p2
output intersect_p1p2 := abs(m_v - m_{l_1}) > ϵ
output intersect_lat_p1p2
  := if is_fnc ∧ intersect_p1p2 then (b_v - b_{l_1}) / (m_{l_1} - m_v) else lat
output intersect_lon_p1p2 := m_{l_1} * intersect_lat_p1p2 + b_{l_1}
trigger intersect_p1p2
  ∧ ((intersect_lat_p1p2 > min_lat_v ∧ intersect_lat_p1p2 < max_lat_v)
  ∧ (intersect_lon_p1p2 > min_lon_v ∧ intersect_lon_p1p2 < max_lon_v))
  ∧ ((intersect_lat_p1p2 > min(lat_{p_1}, lat_{p_2})
   ∧ intersect_lat_p1p2 < max(lat_{p_1}, lat_{p_2}))
  ∧ (intersect_lon_p1p2 > min(lon_{p_1}, lon_{p_2})
   ∧ intersect_lon_p0p1 < max(lon_{p_1}, lon_{p_2})))
  "VIOLATION: line crossing between p1 and p2"
```

Fig. 9: Geo-fencing

A.3 Sensor Cross-Validation

In this section, we consider two complementary sensors. The GPS receiver based on satellite signals and an Inertial Measurement Unit (IMU) based on acceleration and angular rate data. In the specification given in Fig. 10, we use the fact that the acceleration is the derivative of the velocity. This correlation should be present when comparing IMU and GPS receiver events. Since the IMU frequency is known, we compute the expected velocity, by averaging instead of an integration with the trapezoid abstraction (lines 15-16). This removes undesired noise. The average velocity given by the GPS receiver is computed in a straight forward manner (lines 19–22). A warning is raised if the difference in velocity is greater than 0.5 (line 25).

```
import math

// GPS Velocity Submodule
// Input Streams
input speed_h : Float16 // Horizontal Speed
input speed_v : Float16 // Veritical Speed

// IMU Module
input acc_x : Float32 // acceleration in x
input acc_y : Float32 // acceleration in y
input acc_z : Float32 // acceleration in z

// Cross Validation
// IMU velocity
output acc := sqrt((acc_x * acc_x) + (acc_y * acc_y) + (acc_z *
    acc_z))
output IMU_avg_vel @1Hz := acc.aggregate(over: 10s, using:
    avg).defaults(to:0.0) * 0.01

// GPS velocity
output speed_h_diff : Float32 := cast(speed_h -
    speed_h.offset(by:-1).defaults(to:speed_h))
output speed_v_diff : Float32 := cast(speed_v -
    speed_v.offset(by:-1).defaults(to:speed_v))
output all_speed := sqrt(speed_h_diff * speed_h_diff + speed_v_diff *
    speed_v_diff)
output gps_avg_vel @1Hz := all_speed.aggregate(over: 10s, using:
    avg).defaults(to:0.0)

//Comparison
trigger abs(gps_avg_vel - IMU_avg_vel) > 0.5 "VIOLATION: Cross
    Validation."
```

Fig. 10: Cross-validation