A ROS Adapter for RTLola

Jan Baumeister[2], Bernd Finkbeiner[2], Franz Jünger[1], Florian Kohn[2], Sebastian Schirmer[1], and Christoph Torens[1]

[1] German Aerospace Center (DLR), Germany
{franz.juenger, sebastian.schirmer, christoph.torens}@dlr.de
[2] CISPA Helmholtz Center for Information Security
{jan.baumeister, finkbeiner, florian.kohn}@cispa.de

Abstract. This paper presents an adapter for RTLola that simplifies the integration of runtime verification into Robot Operating System (ROS) applications. While ROS is the standard middleware for robotic development, ensuring the safety and reliability of high-level tasks such as navigation and object recognition remains challenging. The adapter facilitates the use of RTLola, a stream-based specification language for defining complex real-time properties, by automatically connecting a generated RTLola monitor to ROS topics and services. As a use case, the adapter was deployed onboard of an unmanned aircraft to reduce false positives in detecting people near the landing site. Specifically, our RTLola monitor cross-validates machine-learning-based person detections against objects in LiDAR depth images using classical computer vision techniques. This experiment demonstrates that runtime verification improves robotic safety while requiring minimal integration effort.

Keywords: Runtime Verification · Stream-based Monitoring · ROS

1 Introduction

We present an adapter that simplifies the integration of RTLola into robotic applications running within the Robot Operating System (ROS). ROS is an open-source framework that provides essential middleware functionality and abstractions to manage the complexity of scaling autonomous robotic missions. For instance, ROS is designed with distributed computing in mind, where essential robotic functions are executed in *nodes* that receive data from other nodes and pass their results along. Such an essential function can be low-level as reading sensors but also more high-level such as navigation and object recognition. For a safe execution, ROS has basic monitoring capabilities. While sufficient for low-level tasks, these capabilities are too limited to ensure the safety of high-level autonomous functions, which often require monitoring complex temporal system properties with asynchronous inputs. As a result, users are typically forced to implement custom monitoring solutions using general-purpose code, increasing the likelihood of errors. To mitigate this risk, runtime verification (RV) offers a formal, lightweight alternative where complex properties are specified concisely

using a formal language, from which executable monitors are automatically generated. RTLola is one such specification language. Based on stream computations, language features natively handle real-time and asynchronicity.

In this paper, we present a ROS adapter for RTLola[1]. This adapter enables seamless integration of a monitor generated with the RTLola framework [1] by wrapping it as a ROS node, eliminating the need for any manual adjustments. The key contributions of the paper are:

- *automatic input mapping*: inspecting the RTLola specification and automatically subscribing to ROS topics to match required streams;
- *automatic output mapping*: inspecting the RTLola ROS topic and automatically publish corresponding streams;
- *supports ROS service*: inspection of RTLola specification and stream mapping for direct request-reply communication;
- *implemented in Rust*: safe and efficient, compatible with any ROS 2 version;
- *tool validated in flight*: used for cross-validation of bounding boxes given by machine-learning-component and blobs detected in LiDAR image using traditional computer vision techniques.

Alongside the technical details presented in the subsequent sections, we also provide information on the experimental flight test, starting with the use-case.

Experimental Use-case We deployed the adapter for safeguarding a machine-learning (ML) component running onboard of an unmanned aircraft. This ML component is responsible for detecting people near a designated landing site. Figure 1 illustrates the scenario, where mannequins simulate people standing on a vertiport – the intended landing site for the hovering unmanned aircraft. Figure 2 depicts the unmanned aircraft equipped with a camera and a LiDAR. In the background, the "container city" is shown. Some mannequins were positioned within this city to provide occlusion scenarios. The aircraft's perspective is shown in Figure 3, highlighting both correct detections (green) and challenging conditions that may lead to false positives (red) of the ML component. In

[1] `https://github.com/DLR-FT/RTLola-ROS2-Adapter`

Fig. 1. Task: Safeguard detecting people at landing site.

Fig. 2. Compare detected persons with objects in a LiDAR depth image.

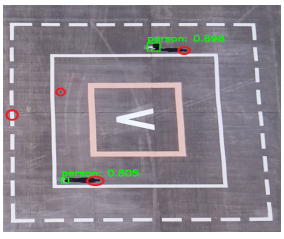

Fig. 3. True positives are shown in green, potential false positives in red.

this experiment, we use the adapter to integrate an RTLola monitor that cross-validates ML-based detections against objects in LiDAR depth images using a traditional Blob detector. In the following, we focus on the monitor integration, while referring to [7] for the experimental results where also other properties such as geofencing were considered for monitoring the operational design domain.

1.1 Related Work

In recent years, the integration of RV into ROS systems has received growing attention. ROS is widely used not only for developing robotic applications but also for supporting various simulation frameworks such as Gazebo [4] or Carla [3], as well as visualization tools like PlotJuggler. Several RV frameworks have been developed to support ROS. For example, RTAMT [8] integrates monitors based on Signal Temporal Logic into ROS applications. The Ogma [5] tool runs monitors specified in languages such as CoPilot, FRET, and Lustre as ROS nodes. Similarily, [2] uses the stream-based specification language TeSSLa to integrate monitors into ROS systems. Similar to them, we introduce the specification language RTLola to ROS. However, unlike prior tools, our tool supports not only topic-based communication, i.e., publishing and subscribing, but also introduces an RTLola service interface. This service enables request-response communication, thereby reducing the need for manual handling of monitoring responses using topic subscriptions. Although ROSMonitoring 2.0 [6] supports services by wrapping monitors in Python, we chose Rust for our adapter due to its advantages in safety and performance. Moreover, we refer to our tool as an "adapter" rather than a "bridge" because it incorporates logic that automatically derives the mappings between ROS and RTLola, offering more intelligent and robust integration.

2 RTLola Specification Language

In stream-based specification languages such as RTLola `input` streams capture observations of the system as a discrete sequence of measurements. `Output` streams aggregate, combine, and transform input streams to compute valuable statistics about the system. Special output streams called `trigger` streams can be declared to specify a Boolean verdict about the systems health based on these statistics. Output streams can be generalized to sets of stream instances that are uniquely identified by a set of parameters. Such parameterized output streams include a `spawn` and `close` condition, specifying when and how new stream instances are created and removed.

Experimental Setup The specification used in the experiments[3] is given in Listing 1.1. The specification tests whether a blob exists within a given bounding box, which should be the case for an actual person. First, in Line 2, four input streams are specified that characterize a single bounding box in the camera image by the `x` and `y` coordinate of its upper left corner and its width and height. Similarly, the input streams `blob_x`, `blob_y`, and `blob_r` describe the `xy`-position and the radius `r` of a single circular blob in the LiDAR image. Since the images of the camera and the LiDAR have different resolutions and only partially overlap w.r.t. their field of views, the two output streams `blob_x_in_cam` and `blob_y_in_cam` transform the coordinates of the detected blob in the LiDAR image into corresponding coordinates in the camera image. Note that due to lens curvature, the transformation is inherently non-linear. Furthermore, the solution depicted here represents a sub-optimal solution that is valid only within a specific altitude band. The next output stream in Line 10 is parameterized over the `x` and `y` coordinates of blobs and their radii `r`, specified by the parameter list followed after its stream name. The `spawn` condition of the stream (Line 11) instantiates these parameters with the (transformed) coordinate and radius of a blob, effectively creating an instance of the stream for each known blob. Each stream instance then validates whether the current bounding box contains the blob the stream instance corresponds to, specified in the `eval` clause of the stream (Line 12 to 13). Since memorizing every blob received at run-time is infeasible, we use the `close` condition: a stream instance (and with that a blob) becomes irrelevant after 200 milliseconds, using the instance's local clock starting upon creation (Line 14). Lastly, the `validate` stream in Line 16 tests whether the current bounding box intersects with any blob by aggregating over all instances of the `check` stream[4]. If such an instance exists, the stream evaluates to true.

```
1  // Service inputs: incoming bounding boxes
2  input bb_x, bb_y, bb_w, bb_h: Float64, Float64, Float64, Float64
3  // Subscribed inputs: detected blobs in depth image
4  input blob_x, blob_y, blob_r: Float64, Float64, Float64
5  // Transform depth image coordinates to camera coordinates
6  output blob_x_in_cam := -0.01*blob_y**3.0 + 1.81*blob_y**2.0
7                          -66.24*blob_y + 1765.75
8  output blob_y_in_cam := 13.91*blob_x - 6627.13
9  // Whenever a new bounding box arrives compare it to all recent blobs
10 output check(x, y, r)
11    spawn with (blob_x_in_cam, blob_y_in_cam, blob_r)
12    eval with x - r > bb_x ∧ x + r < bb_x + bb_w ∧
13              y - r > bb_y ∧ y + r < bb_y + bb_h
14    close @Local(200ms) // recent is defined as 200ms
15 // Service response
16 output validate := check.aggregate(over_instances: all, using: ∃)
```

Listing 1.1. An RTLola specification matching bounding boxes to blobs.

[3] The specification together with a toy trace can be tested within a playground: https://rtlola.cispa.de/playground/tutorial

[4] In practice, bounding boxes are sent with a delay to not only consider past blobs.

3 ROS 2 Environment

ROS is a middleware that enables the execution of a distributed system made up of nodes, where each *node* typically handles a specific task in the robotic application. For example, one node may handle a sensor driver that collects sensor readings, another node may manage control tasks, and a third node may focus on monitoring. ROS supports different ways of communication between nodes. Most prominent are topics and services.

A *topic* is a named communication channel over which structured messages of a specific type are sent. To send data on a topic, a node must *publish* messages to it, while to receive, a node must *subscribe* to the topic. Multiple nodes can simultaneously publish to the same topic, and multiple nodes can subscribe. Each topic is strictly associated with one message type, and all publishers and subscribers must agree on this type. Listing 1.2 provides the interface file for the "blob" topic published by the Blob detector: a blob's `xy`-position and radius.

A *service* is a named communication channel that allows one node to request a specific response from another node. To initiate a service, a node must send a *request* message (client), while the node providing the service must process the request and sends a *response* (server). Each service is associated with a request and response message type, which must be defined beforehand, ensuring both the client and server nodes agree. Unlike topics, services operate in a one-to-one manner. A service call is blocking, meaning the client waits for a response before continuing. Listing 1.3 provides the interface file for the RTLola service that validates bounding boxes. The request and the response are separated by "`---`": the request specifies the `xy`-position, width, and height of the bounding box, while the response returns a single Boolean which indicates a matching blob.

Quality of Service (QoS) defines how data is exchanged between nodes. Some example settings include the *history* policy, which determines whether to keep only a limited number of recent messages ("keep last") or all messages ("keep all"), with the *queue size* applying only to the former; *reliability*, which defines whether messages are delivered with possible loss ("best effort") or guaranteed with retries ("reliable"); and *durability*, which controls if messages are persisted for late subscribers ("transient local") or discarded after publishing ("volatile").

ROS primarily support C++ and Python. C++ is widely used for performance-critical and hardware-related tasks, while Python is popular for scripting, rapid prototyping, and creating simpler nodes. Therefore, most core ROS components are written in C++. ROS comes with its own build system using CMake/Catkin, which makes the use of other compiling languages building on other build systems challenging, e.g., Rust using Cargo.

```
float64 x // x-position
float64 y // y-position
float64 r // radius
```

Listing 1.2. "blob" interface file

```
float64 x // x-position
float64 y // y-position
float64 w // width
float64 h // height
--- // Separates request and response
bool validate // is true positive
```

Listing 1.3. "bb" service interface file

Experimental Setup The unmanned aircraft was a hexacopter with a maximal takeoff weight of 15.5 kg that followed a preprogrammed waypoint mission flying above the mannequins. To guarantee safety, a human remote pilot had always the possibility to takeover control when observing unintended drone behavior. As hardware payload, the drone carries a Pixhawk 4 flight controller hardware, a Jetson jAi Go 2400 camera, an Ouster OS0 LiDAR, and a Nvidia Jetson AGX Orin companion computer. The companion computer is connected via 5 GHz Wlan to a ground control station for controlling the experiment and visualizing the validated bounding boxes.

Software-wise, the Pixhawk 4 runs the PX4 autopilot software and the Nvidia Jetson executes a ROS environment with multiple nodes. As PX4 and ROS provide a deep integration that directly allows to exchange information[5] numerous PX4 messages are directly available as ROS topics for the companion computer. This includes for instance information about battery, actuators, and sensor readings such as position[6]. The central ROS nodes for this experiments running on the companion computer were a node that publishes the camera images, a node for publishing the LiDAR depth images, a node that runs a OpenCV Blob detector on the depth images, an RTLola monitor, and DLR's UAVision that runs a ML-based object detector and streams verified detections to a ground control station. The ML-component ran at ∼3 Hz and the Blob detector ran at around ∼20 Hz. UAVision is subscribed to the images from the camera publisher. When UAVision receives a camera image, it first runs the ML-based detector to obtain bounding boxes. It then uses a service provided by the RTLola monitor node to send each detected bounding box for cross-validation with the Blob detections subscribed to by the monitor If such a corresponding blob exists, the bounding box is validated; otherwise, it is falsified. This information, along with the bounding box, is transmitted to the ground control station. Validated bounding boxes are depicted in cyan, while falsified ones are shown in pink.

4 A ROS Adapter for RTLola

The adapter is written in Rust to ensure seamless integration with RTLola. It assumes a running ROS 2 workspace. To build and integrate the monitor for a given RTLola specification, the user simply needs to execute:

`cargo run - <specification>`.

Figure 4 shows the corresponding pipeline stages. During the "Generate" stage, the adapter collects information about available topics using ROS command-line tools. It starts by executing `ros2 topic list` to retrieve a list of currently available topics. Then, `ros2 interface show <topic/service>` is used to obtain the corresponding interface definitions. Finally, QoS details are accessed via `ros2 topic info <topic/service> -verbose`. This gathered information is used to generate Rust code instances of pre-defined templates, enabling topic subscrip-

[5] `https://docs.px4.io/main/en/ros2/user_guide.html`
[6] `https://github.com/PX4/px4_msgs`

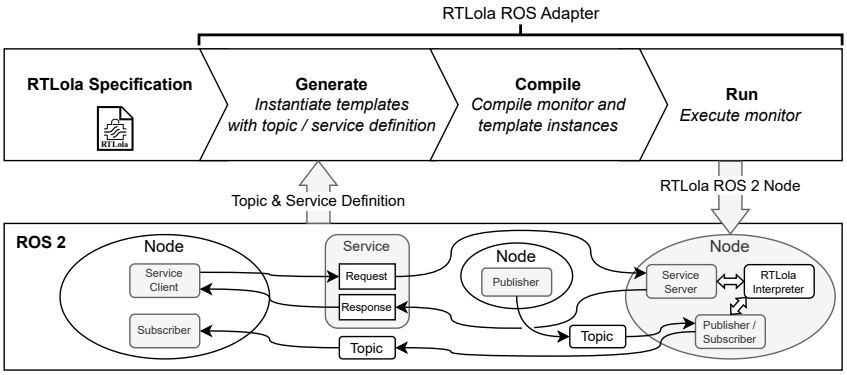

Fig. 4. Execution pipeline of the ROS adapter for RTLola

tion. These templates utilize the Rust crate `r2r`[7], which avoids the integration with the ROS build system and instead relies solely on `cargo`. By default, the subscriptions matches the QoS settings of the publisher to ensure compability. Whether the monitor publishes or provides a service depends on the presence of the `RTLolaOutput` topic and the `RTLolaService` service interface in the running ROS workspace, respectively. If either interface is available, corresponding templates are instantiated to implement a publisher or a service server. Note that this stage is independent of the provided RTLola specification – the mapping of input/output RTLola streams are derived in a later stage. Further, a configuration file allows for QoS customization and includes a flag to "freeze" this stage's output, preventing overwriting of already generated code. During the next stage, the generated template instances and the monitor code are compiled. Finally, the wrapped RTLola monitor is executed as ROS node. An executed RTLola monitor first generates a mapping between RTLola input/output streams and corresponding topic members. This mapping adheres to a naming convention: <*package_name_lowercase*>_<*member*>, which is also applied during template instantiation. For example, the input stream `input adc_a0: Float32` maps to the float member `a0` of topic `adc`. Moreover, fixed-length arrays in topics are unrolled, e.g., a topic *Gps* with member `float64[2] x` results in two streams: `gps_x_0` and `gps_x_1`. If any required stream is missing, the adapter will raise an error to notify the user. Note that the adapter only needs to be recompiled when topics or services are added or modified. Limitations: When responding to a service request, the adapter uses the immediate output stream values after processing the inputs. As a consequence, outputs of periodic streams may remain unchanged in the response. Additionally, topic members of unbounded array types are currently not supported and are therefore ignored.

[7] `https://github.com/sequenceplanner/r2r`

Experimental Results Prior to the flight test, the adapter was used during ground tests. The code generated during these tests was frozen by setting the respective flag to avoid code generation during flight tests. The PX4 autopilot provides over 50 topics. The adapter generated code for all necessary topics, so the user can focus fully on writing the specification while required topics are subscribed automatically

The provided RTLola service simplified the implementation of the cross-validation called by UAVision. If these requests were implemented using topics, additional boilerplate code would be required to handle the request-response pattern manually. Using a service avoids this overhead. Listing 1.4 shows a code snippet demonstrating how to integrate the service. Line 4 and 5 retrieve an image, detect persons, and create requests. For each detected bounding box, a blocking request `req_bb` is sent and handled asynchronously using a "future" response (Line 7). This response, once available, is passed to a callback function, responsible for forwarding the result to the ground control station (Line 8).

```
def callback(request, response): # callback function
  ...
while True:
  frame = self.camera.read() # reads camera frame
  req_bbs = self.ml.detect(frame) # detection and creation of requests
  for req_bb in req_bbs: # iterates over bounding box requests
    self.ros_client.call_async(req_bb).add_done_callback(
      functools.partial(self.callback, req_bb)) # bind request to callback
```

Listing 1.4. Excerpt of Python code demonstrating how the RTLola service simplifies monitor integration into UAVision, avoiding explicit subscription handling.

The experiment also demonstrates that runtime verification improves the system's performance. Figure 5 illustrates a scenario where detections were successfully validated and could therefore be trusted. In contrast, Figure 6 shows an example of a false positive detection that was identified as untrustworthy and subsequently discarded by the system.

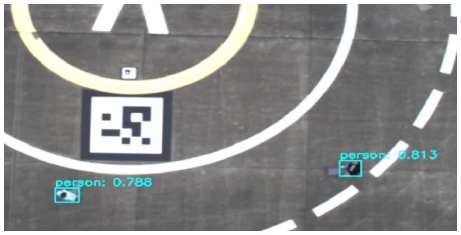

Fig. 5. The monitor validated two correct human bounding boxes (cyan).

Fig. 6. Monitor flags a false positive (pink) with no matching blob.

5 Conclusion

We presented a ROS adapter for RTLola, designed to simplify the integration of RTLola into ROS-based applications by automatically mapping ROS topics to RTLola streams. In real-flight experiments, the adapter successfully generated code for all required topics, significantly reducing user effort given the large number of available topics. Further, we demonstrated that providing a service interface by the monitor simplifies the usage of monitoring verdicts by eliminating manual request-response handling. In future, we plan to advance the cross-validation specification. Instead of checking if a blob is within a bounding box, we plan to test other metrics such as the Hausdorff distance or explicit overlap computation – thereby improving the accuracy of the cross-validation.

Acknowledgments

This work was partially supported by the German federal aviation research program (LuFo ID: 20D2111C, ID: 20Q1963B, and ID: 20Q1963C), German Research Foundation (DFG) as part of TRR 248 (No. 389792660), and by the European Research Council (ERC) Grant HYPER (No. 101055412). Sebastian Schirmer carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

References

1. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. CoRR **abs/2501.15913** (2025). https://doi.org/10.48550/ARXIV.2501.15913, https://doi.org/10.48550/arXiv.2501.15913
2. Begemann, M.J., Kallwies, H., Leucker, M., Schmitz, M.: TeSSla-ROS-Bridge - Runtime Verification of Robotic Systems. In: Ábrahám, E., Dubslaff, C., Tarifa, S.L.T. (eds.) Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14446, pp. 388–398. Springer (2023). https://doi.org/10.1007/978-3-031-47963-2_23, https://doi.org/10.1007/978-3-031-47963-2_23
3. Dosovitskiy, A., Ros, G., Codevilla, F., López, A.M., Koltun, V.: CARLA: an open urban driving simulator. In: 1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings. Proceedings of Machine Learning Research, vol. 78, pp. 1–16. PMLR (2017), http://proceedings.mlr.press/v78/dosovitskiy17a.html
4. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566). vol. 3, pp. 2149–2154 vol.3 (2004). https://doi.org/10.1109/IROS.2004.1389727
5. Perez, I., Mavridou, A., Pressburger, T., Will, A., Martin, P.J.: Monitoring ROS2: from requirements to autonomous robots. In: Luckcuck, M., Farrell, M. (eds.) Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022,

and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE)Berlin, Germany, 26th and 27th of September 2022. EPTCS, vol. 371, pp. 208–216 (2022). https://doi.org/10.4204/EPTCS.371.15, https://doi.org/10.4204/EPTCS.371.15

6. Saadat, M.G., Ferrando, A., Dennis, L.A., Fisher, M.: ROSMonitoring 2.0: Extending ROS Runtime Verification to Services and ordered topics. In: Luckcuck, M., Xu, M. (eds.) Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024. EPTCS, vol. 411, pp. 38–55 (2024). https://doi.org/10.4204/EPTCS.411.3, https://doi.org/10.4204/EPTCS.411.3
7. Torens, C.: Runtime-monitoring of operational design domain to safeguard machine learning components. Special Issue HorizonUAM, under review. CEAS Aeronaut. J (2023)
8. Yamaguchi, T., Hoxha, B., Ničković, D.: RTAMT – Runtime Robustness Monitors with Application to CPS and Robotics. International Journal on Software Tools for Technology Transfer **26**(1), 79–99 (Oct 2023). https://doi.org/10.1007/s10009-023-00720-3, http://dx.doi.org/10.1007/S10009-023-00720-3