ROSMonitoring: a Runtime Verification Framework for ROS*

Angelo Ferrando[1], Rafael C. Cardoso[1], Michael Fisher[1], Davide Ancona[2], Luca Franceschini[2], and Viviana Mascardi[2]

[1] Department of Computer Science, University of Liverpool, Liverpool, United Kingdom {angelo.ferrando, rafael.cardoso, mfisher}@liverpool.ac.uk
[2] Department of Computer Science, Bioengineering, Robotics and Systems Engineering (DIBRIS), University of Genova, Genova, Italy
luca.franceschini@dibris.unige.it, {davide.ancona, viviana.mascardi}@unige.it

Abstract. Recently, robotic applications have been seeing widespread use across industry, often tackling safety-critical scenarios where software reliability is paramount. These scenarios often have unpredictable environments and, therefore, it is crucial to be able to provide assurances about the system at runtime. In this paper, we introduce ROSMonitoring, a framework to support Runtime Verification (RV) of robotic applications developed using the Robot Operating System (ROS). The main advantages of ROSMonitoring compared to the state of the art are its portability across multiple ROS distributions and its agnosticism w.r.t. the specification formalism. We describe the architecture behind ROSMonitoring and show how it can be used in a traditional ROS example. To better evaluate our approach, we apply it to a practical example using a simulation of the Mars curiosity rover. Finally, we report the results of some experiments to check how well our framework scales.

1 Introduction

There are many different techniques that can be used to provide reliability assurances for software applications. Formal verification allows us to verify the correctness of a software application against some kind of formal logic specification, for example Linear Temporal Logic (LTL [12]). Using formal verification, the system can be analysed at design time (offline) and/or at runtime (online).

Verification at design time, such as model checking [13], exhaustively checks the behaviour of a system. This is done by generating a formal model of it and then performing a state space search looking for the satisfaction of the formal properties in all possible executions.

Runtime verification (RV [10]) is a more lightweight approach which is usually more suitable for examining "black box" software components. RV focuses on

* Work supported by the UK Research and Innovation Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE), EP/R026173 (ORCA), and EP/R026084 (RAIN).

analysing only what the system produces while it is being executed and, because of this, it can only conclude the satisfaction/violation of properties regarding the current observed execution. Since RV does not need to exhaustively check the system behaviour, it scales better to real systems, since it does not suffer from state space explosion problems that can be commonly found in model checking.

There are many techniques and implementations of RV that use different formalisms, such as LTL [4] or finite automata over finite and infinite strings [11]. One of the most common approaches to perform RV of a system is through monitoring, where monitors are used to check the system execution against formally specified properties. This check can happen incrementally at runtime (online RV), or over recorded executions such as log files (offline RV).

Even though many frameworks and libraries exist that support RV of software systems developed across many different programming languages, there are only a few suitable for monitoring robotic applications in ROS. One of the most promising is ROSRV [9], a general-purpose runtime verification framework for ROS. ROSRV uses the Monitoring-Oriented Programming (MOP [6]) paradigm. However, one of the main drawbacks of ROSRV is that it has very limited portability, in fact, the latest version available at the time of writing can only be used with the ROS *Groovy Galapagos* distribution, which stopped being supported in 2014. This lack of portability in such dynamic and evolving context leaves a significant gap, and it is one of the main motivations for developing our new framework, called ROSMonitoring.

The ROSMonitoring RV is a general, formalism agnostic (i.e. does not depend on using only a specific verification formalism to represent and check properties), runtime monitoring framework that can be used with multiple ROS distributions (tested in Melodic and Kinetic). It is designed for the automatic verification of the communication between ROS nodes by monitoring topics and checking against formal properties expressed using the user's formalism of choice. Because of this, ROSMonitoring can be applied to any kind of ROS-based robotic application, with no limitation on how each communication endpoint is implemented.

2 ROSMonitoring

ROSMonitoring[3] is a framework for runtime monitoring of ROS topics, and to do so it creates monitors that are placed between ROS nodes to intercept messages on relevant topics and check the events generated by these messages against formally specified properties.

Our framework has three main components: (a) *instrumentation*, used for automatically creating a monitor and inserting it in the middle of the communication among ROS nodes; (b) *oracle*, used for checking whether the events that are observed by the monitor conform to some formal specification or not; and (c) *monitor*, the implementation of the ROS monitor, it is responsible for intercepting messages between nodes and communicating with the oracle.

[3] `https://github.com/autonomy-and-verification-uol/ROSMonitoring`

In Figure 1, we provide a high-level overview of ROSMonitoring. From left to right: we create the monitor nodes and perform node instrumentation according to a YAML configuration file customised by the user; as output we obtain the new instrumented nodes (with the communication gaps), and the ROS monitor implementation as a Python node; and we run ROS with the instrumented nodes and the monitor nodes, which performs RV either online or offline, depending on the configuration file used.

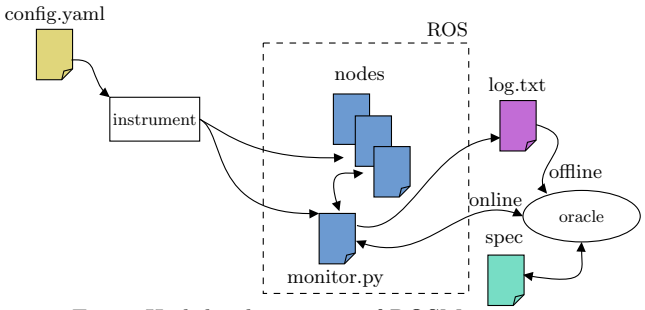

Fig. 1: High-level overview of ROSMonitoring.

2.1 Instrumentation

First, let us consider a common publisher/subscriber example that can be found in ROS tutorials. The example comprises two nodes communicating on a specific topic. One node is called *talker*, and continuously publishes the message `"hello"` in the topic *chatter*. The *listener* node subscribes to the *chatter* topic.

We use a YAML configuration file to guide the instrumentation process. ROS users should be familiar with YAML, as it is also used for configuration there. Within this file, users can set the number of needed monitors, and then for each monitor can set the topics that have to be intercepted, including: the name of the topic, the ROS message type that is expected in that topic, the type of action that the monitor should perform, which side (publisher or subscriber) is the monitor intercepting the message, and the names and launch path of the nodes that are going to be remapped. After preferences have been configured in *config.yaml*, the last step is to run the *generator* script to automatically generate the monitors and instrument the required ROS launch files. By default, the monitors so generated can perform two different actions when intercepting a message: *log*, the monitor will simply log the observed events into the log file specified in the monitor's `log` attribute; and *filter*, the monitor will filter out all the events inconsistent with the specification (requires an oracle).

For instance, in Listing 1 we have the configuration file for the chatter example. Note that we need a unique id for each monitor, and that the monitors always log the events received (thus, we also need to specify a log file for each of them). In this example, the list of topics we want to intercept only contains *chatter*. In general, we can have more topics that we want to monitor in our application, and with this field we are able to add as many as necessary. It is important that the ROS message type is exactly as it is specified in ROS (i.e.

ROS is able to find the given message type), in this case it is the primitive ROS message type *String*.

```
monitors:
  - monitor:
    id: monitor_0
    log: ./log_0.txt
    topics:
      - name: chatter
        type: std_msgs.msg.String
        action: filter
        side: publisher
          - node: talker
            path: /chatter/launch/chatter.launch
```

Listing 1: Configuration file for the chatter example.

By passing the path to the file that contains the launch code for the *talker* node we can automatically instrument the node by remapping the *chatter* topic to *chatter_mon*. Note that this is done only for the specified side, the publisher (*talker*) node in this example. The following is automatically added to the appropriate section of the *chatter.launch* file:

<remap from="chatter" to="chatter_mon"/>

Remapping topic names allows us to create a gap in communication. In this example, the two nodes are now unable to communicate directly anymore, because the instrumented *talker* publishes on a different topic from the one subscribed by the *listener*. We fill this gap by adding our monitor, which is automatically generated during the instrumentation step. The monitor subscribes to *chatter_mon* and publishes to *chatter*. Then, the monitor has to decide if it wants to propagate the message or not with the help of an oracle, further described in Section 2.2.

There are some applications where this type of invasive monitoring is not desired, such as when it is not required to intercept incorrect messages (e.g. in offline RV), or when dealing with proprietary code. In such cases, we can omit the last part of the configuration file (side, node, and path). By doing so, the monitor no longer intercepts messages between ROS nodes, but it still has access to the relevant messages and can log them or do another appropriate action.

2.2 Oracle

One of ROSMonitoring aims is to be highly reusable. The best way for achieving this result is being formalism agnostic, and we obtain this by introducing the notion of oracle. The oracle is an external component, which can be produced by third-parties, containing the logic and implementation of a formalism.

```
oracle:
  port: 8080
  url: 127.0.0.1
```

Listing 2: Extra configuration lines for adding an oracle.

The oracle must be listening on a specific URL (specified in the *url* attribute) and port (specified in the *port* attribute). These attributes are specified in the instrumentation configuration file. If we wanted to add an oracle to the previous example from Listing 1, then we would add the snippet in Listing 2 between the *log* and *topics* attributes.

In the offline scenario, the oracle is used for checking the log file produced by the ROS monitor node. In the online scenario, each time an event is intercepted by the ROS monitor, the latter is propagated to the oracle, which has to check it and to reply with a verdict accordingly to the current satisfaction or violation of some formal property (specified in the oracle side using the formalism supported by the oracle). Upon the reception of this verdict, the ROS monitor decides what to do with the event.

ROSMonitoring requires very few constraints for adding a new oracle. We use *JSON* as data-interchange format for serialising the messages observed by the ROS monitor. The ROS messages are first translated into their JSON representation, which are then logged (offline case) or sent to the oracle (online case). The oracle must be listening and ready to receive the JSON messages using the *WebSocket* protocol. We chose WebSocket because of its real-time bi-directional point to point communication between client and server, which does not require a continue polling of the server. For demonstration purposes, we describe two default oracles available.

Runtime Monitoring Language (RML) Oracle. This oracle supports RML[4] for the specification of properties. RML is a rewriting-based and system agnostic Domain Specific Language (DSL) for RV used for defining formal properties and synthesising monitors from them; it is inspired by trace expressions [2], a formalism adopted for RV in several contexts, such as Multi-Agent Systems (MAS) [8], Object-Oriented programming [7], and Remote Patient Monitoring (RPM) [3]. We chose RML because it imposes no restrictions on how events are generated, and is completely system agnostic. Thanks to this, RML monitors and specifications can be reused in many different contexts, and since RML is based on the data-interchange format JSON for representing events, its synthesised monitors can easily interoperate with other JSON compatible systems, such as ROSMonitoring. The RML oracle is implemented in SWI-Prolog, along with the event calculus on which the semantics of RML is based. Given an RML specification, it is first compiled into its intermediate language representation implemented in SWI-Prolog (trace expressions [2]), then the SWI-Prolog oracle starts listening on a WebSocket for checking incoming JSON events intercepted by the ROSMonitoring monitor.

Reelay Oracle. Reelay[5] is a header-only C++ library and set of tools for system-level verification and testing of real-time systems. Reelay implements state-of-the-art runtime verification techniques to construct runtime monitors that check temporal behaviours of the system against system-level requirements.

[4] `https://rmlatdibris.github.io/`
[5] `https://doganulus.github.io/reelay/`

Reelay supports definition of temporal properties, extended with past operators, such as LTL, MTL, and STL (Linear, Metric and Signal Temporal Logic respectively). Since the Reelay library does not expect JSON messages as input to the monitors, we integrated the latter inside a Python oracle implementation which takes care of this message translation.

2.3 The ROS monitor

When the instrumentation program analyses our configuration file, in addition to the resulting instrumentation launch files, it also produces the ROS monitor code. Each monitor is automatically generated into a ROS node in Python, which is a native language supported in ROS. By default, the monitor can log or filter the intercepted messages accordingly to the configuration chosen in the instrumentation step.

ROSMonitoring always logs the events generated by the messages in the monitored topics. The best option if we are only concerned in logging messages and not intercepting them, is to employ an external monitor (i.e. not interfering with the messages).

There is an extra configuration attribute that can be set during instrumentation called *warning*. Warning is a flag that determines when the monitor should publish a warning message containing as much information as possible about a property that has been violated. This message is published on a special topic called *monitor_error* and has its own message type *MonitorError.msg*. Information in this message includes: the topic that originated the event, the content of the message that was intercepted in that topic, the property that was violated, and the ROS time of when the message was intercepted.

The algorithm for automatically generating the monitors is fairly straightforward. We report the pseudo-code for the offline (Algorithm 1) and online (Algorithm 2) monitors that would be generated for a chosen set of topics T_1, ..., T_n. In the offline scenario, we first subscribe to the topics we want to keep track of. Each time a message is received the callback in lines 6–9 is activated and the ROS message is translated to JSON. This translation can be easily achieved using the *rospy_message_converter* package. We assume that the oracle can read JSON messages, a fairly common message format. Then, the converted message is logged, ready to be used by an oracle when the system is offline.

Algorithm 1 Offline monitor generated for topics $T_1 ... T_n$.

```
1: function OFFLINE_MONITOR
2:     for i = 1 to n do
3:         CREATE_SUBSCRIBER(T_i, receive_msg)
4:     end for
5: end function
6: function RECEIVE_MSG(ros_msg)
7:     json_msg = CONVERT_ROS_MSG_TO_JSON(ros_msg)
8:     LOG(json_msg)
9: end function
```

In Algorithm 2, we create a publisher for each topic (just *chatter* in our example) and a subscriber for each instrumented version (*chatter_mon* in our

example). When the subscriber is created, we pass the corresponding publisher that will be used to eventually republish the message. Then, we have two callback functions that are activated when receiving messages. The first callback (lines 7–10) is called upon the reception of a ROS message on an instrumented topic. Inside this callback, the ROS message is first translated to JSON and then propagated to the oracle. On line 9 we also inform which callback has to be called when the response from the oracle arrives (`oracle_msg`).

Algorithm 2 Online monitor generated for topics $T_1 ... T_n$.

```
1: function ONLINE_MONITOR
2:     for i = 1 to n do
3:         pub_T_i = CREATE_PUBLISHER(T_i)
4:         CREATE_SUBSCRIBER(T_i_mon, receive_msg, pub_T_i)
5:     end for
6: end function
7: function RECEIVE_MSG(ros_msg, pub)
8:     json_msg = CONVERT_ROS_MSG_TO_JSON(ros_msg)
9:     SEND_TO_ORACLE(json_msg, oracle_msg, pub)
10: end function
11: function ORACLE_MSG(res, pub)
12:     verdict = EXTRACT_VERDICT(res)
13:     json_msg = EXTRACT_JSON_MSG(res)
14:     ros_msg = CONVERT_JSON_TO_ROS_MSG(json_msg)
15:     if verdict then
16:         pub.PUBLISH(ros_msg)
17:     end if
18:     LOG(json_msg)
19: end function
```

The `oracle_msg` callback (lines 11–19) extracts the verdict and the JSON message from the oracle's reply. Next, it converts the JSON message back to a ROS message. This is needed if the oracle can change the contents of the message in any way, such as when it is equipped to perform failure handling. Then, if the verdict is valid we republish the message on the corresponding topic name.

After receiving the response from the oracle, the first thing the monitor does is to check for the presence of the attribute *'error'* inside the JSON answer. This is used by the oracle for communicating the validity of the current event. In case of error the event is logged and if the monitor is set to filter wrong messages then it does not propagate the message. When the message is consistent, the monitor simply logs the event and republishes it to the correct topic.

3 Evaluation

We evaluate ROSMonitoring through two separate experiments. First, we apply it to a case study based on a simulation of the Mars curiosity rover. We use this practical example to demonstrate the different features in our framework. In the second experiment, we stress test the delay that can be introduced by adding our monitors in the chatter example.

3.1 Simulation: Mars Curiosity Rover

Curiosity is a rover sent by NASA to explore the surface of Mars. Its main objectives include recording image data and collecting soil/rock data. Although in the original mission the software used in Curiosity was not ROS-based, a ROS version has been developed using official data and 3D models of Curiosity and Mars terrain that were made available by NASA.

We applied our framework to the Curiosity case study by using the filter action to intercept external messages sources (e.g. human or autonomous agent) that violate our property. Due to space constraints and their simplicity, we do not show examples based on the log action, but source code with these examples are available in ROSMonitoring repository.

As an example of the filter action, consider an action library in ROS that controls the wheels of the rover. Action libraries are similar to ROS services, both can receive a request to perform some task and then generate a reply. The difference in using action libraries is that the user can cancel the action, as well as receive feedback about the task execution. ROSMonitoring can only monitor messages that are sent through topics; however, in complex ROS applications it is common to have external commands (e.g. human control for semi-teleoperated movement, or an autonomous agent that sends high-level commands), and these are usually received on a topic that an action client subscribes to.

In this setting, a human sends movement messages to the *wheels_control* topic. The content of the message includes: the *speed* of the wheels, the *direction* for the rover to move (forward, backward, left, or right), and the *distance* that it should move (for how long it should move before it stops). The configuration file for this example is shown in Listing 3.

```
monitors:
  - monitor:
    id: monitor_0
    log: ./log_0.txt
    oracle:
      port: 8080
      url: 127.0.0.1
    topics:
      - name: wheels_control
        type:
        curiosity_mars_rover_description.msg.Move3
        action: filter
        warning: True
        side: subscriber
          - node: wheels_client
            path: /curiosity/launch/wheels.launch
```

Listing 3: Configuration file for the first curiosity example.

Note that since the message is being published by a human (i.e. it is not coming from a ROS node), we have to instrument the subscriber side in this example. Our instrumentation does so by remapping *wheels_control* to *wheels_control_mon* in the launch file of the *wheels_client* node.

Due to the gravity and rocky/difficult terrain in Mars, the Curiosity has to be careful with its speed. Thus, when we intercept a message in the *wheels_control* topic, the message is sent to the oracle to verify the following property:

```
left_speed matches {topic:'wheels_control', direction:'left',
    speed:val} with val <= 10;
right_speed matches {topic:'wheels_control', direction:'right',
    speed:val} with val <= 10;
forward_speed matches {topic:'wheels_control', direction:'forward',
    speed:val} with val <= 15;
backward_speed matches {topic:'wheels_control',
    direction:'backward', speed:val} with val <= 15;
Main = (left_speed \/ right_speed \/ forward_speed \/
    backward_speed)*;
```

That is, if the direction is left or right (i.e. a turn action) then the speed can not be greater than 10, and if the direction is forward or backward then the speed can not be greater then 15. These are arbitrary numbers that were defined based on testing to prevent Curiosity from rolling over. If the verdict from the oracle comes back as an error, then the message is discarded. Otherwise, the message is propagated to the *wheels_control_mon* topic. This filtering monitor correctly prevents any messages that could cause the rover to crash due to a high speed turn.

3.2 Scalability

We stress test ROSMonitoring in a scenario with multiple nodes, multiple topics, and different frequency of messages sent/received per second. The structure of the nodes is very similar to the *chatter* example discussed previously; each node publishes only on one topic and subscribes to the topics on which all the other nodes publish. The goal of each node is to receive a preset number of messages, once received the node can stop. The delay introduced by the presence of the monitor(s) determines the monitor(s) overhead.

In our experiment we set the number of nodes to 10 and varied the frequency rate (i.e. messages published per second). We chose three different values: 100, 500 and 1000 [Hz]; since we have 10 publishers which publish on only one topic each, the total number of topics is 10. Thus, we can reach 1000, 5000 and 10000 messages published per second, respectively. Since we were interested in the overhead introduced by the presence of the monitors, in all the experiments we kept the property to be verified as fixed. More specifically, we chose a property which analyses each event in constant time, and is always considered satisfied. In Figure 2 we show the overhead introduced by the monitor(s) for the different frequency rates; these results have been obtained using ROS Melodic (Version 1.14.3), on a machine with the following specification: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 4 cores 8 threads, 16 GB RAM DDR4.

In the first case, having 10 nodes, 10 topics and a frequency of 100 [Hz], the number of messages sent is 1000 [msg/sec]. For such a low number of messages, the presence of one or multiple monitors is practically transparent to the system. This means that the workload is low enough for the monitor(s) to keep up with the message rate. By increasing the message rate we observe a performance decrease introduced by the monitors. In the second case, with a frequency rate of 500 [Hz], we can observe how with only 1 monitor the system becomes almost

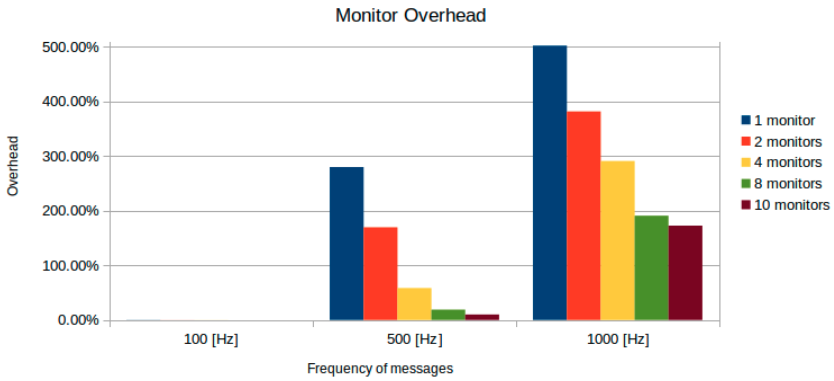

Fig. 2: Overhead introduced by the monitors.

4 times slower (∼270% overhead). This is due to the high number of messages that the monitor has to intercept, 5000 [msg/sec]. The results we obtained for this second case shows the importance of distributing the workload on multiple monitors; in fact, we observe the monitors overhead dropping, reaching ∼9% with 10 monitors (1 monitor for each topic). Naturally, if the number of messages to be handled is too high, even distributing the workload on multiple monitors can not be enough, as we may observe in the third case when the frequency is set to 1000 [Hz] (10000 [msg/sec]).

It is interesting to note that the monitors are more influenced by the higher number of messages to receive, rather than messages to send. The reason for this can be found in the monitor implementation. For each received message, the monitor propagates the information to an external oracle, through web sockets. This communication is sequential in order to preserve the order of the messages that the oracle has to analyse. Increasing the number of messages at this side can cause bottleneck problems, and it is the main reason for the performance decrease. On the other hand, increasing the number of messages to send (propagate) to the other nodes has less influence on performance. We obtain this result because, once the monitor receives the message back from the oracle, it can propagate the latter in parallel exploiting the ROS API.

Besides analysing the monitor's overhead, we evaluated how the presence of monitors delay the communication. As in the overhead analysis, the distribution of the workload on multiple monitors reduces the delay per message, but, when the frequency of messages to intercept increases too much and the monitors become bottlenecks for the communication, the messages start being queued and this causes the increasing delay. We noticed that with a slow frequency (100 [Hz]) the delay is negligible (< 1 [ms]), and can be limited to few seconds (< 5 [sec]) with a high frequency (500 [Hz]). Frequencies higher than that can not be realistically analysed by the monitors.

We also evaluated in our experiments the latency of the error messages. This aspect is mandatory to determine the reaction time of the system in the presence of errors. Also in this case, for reasonable frequencies the reaction time is low,

but when the frequency is too high, the nodes would know about the occurrence of an error with too much delay.

4 Related Work

ROSRV [9] shares some similarities with our framework. Both use monitors, not just for passively observing the messages exchanged among the nodes, but also for intercepting and possibly dealing with incorrect behaviours. The main difference between ROSRV and ROSMonitoring is how they create and insert the monitor in the system. More specifically, ROSRV achieves this by swapping the ROS Master node with a new one, called RVMaster. By replacing the Master node, all node communication has to pass through RVMaster; the latter then establishes peer-to-peer communication by adding the monitor as the man-in-the-middle. In ROSMonitoring we do not change the ROS Master node in any way, instead, we add the monitor through node instrumentation.

In [1] the authors present DeRoS, a DSL for describing safety rules. DeRos provides automatic code generation to integrate these rules with runtime monitoring by generating a ROS safety monitoring node. The logic of the monitor and its integration in ROS are merged. They use the specification to derive the ROS monitor. Unfortunately, this causes a high coupling between the formalism and ROS. In ROSMonitoring instead, the logic of the monitor is external to the low-level integration of the ROS nodes.

Performance Level Profiles (PLP) [5] is an XML-based language for describing the expected properties of functional modules. They provide tools for the automatic generation of code for runtime monitoring of the described properties. Their approach is very close to ours, and goes in the direction of a more general purpose verification process for robotic systems. However, they focus more on a performance checking viewpoint. As in ROSMonitoring, they generate monitors for ROS trying to decouple the monitor's logic from the ROS implementation. Their approach is flexible with respect to the target system (does not have to be only ROS), but requires the specification formalism to be fixed; on the contrary, in ROSMonitoring, the target system is fixed (ROS), but the specification formalism can change.

5 Conclusion

In this paper we presented ROSMonitoring, a new framework for RV of robotic applications in ROS. We showed how, starting from a set of ROS nodes, we can build a ROS monitor node to intercept all the topics related to the properties we want to verify. We described how we implemented ROSMonitoring and its main differences with the state of the art tool ROSRV. The strength of ROSMonitoring lies in its portability and being formalism agnostic, resulting in a framework that is completely decoupled from ROS distribution and oracle implementation.

Future work includes a quantitative comparison against ROSRV, and applying our framework to more practical applications. An interesting extension of

ROSMonitoring would be enriching the information exchanged with the oracle. Instead of only communicating the verdict for a single event, it would be interesting to add the global verdict on the trace. Then, it would be possible to simplify the ROS monitor to automatically republish everything, because, if the monitor knew that the property checked by the oracle has been already satisfied, there is no point in propagating the messages to the oracle anymore.

References

1. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Towards rule-based dynamic safety monitoring for mobile robots. In: SIMPAR. Lecture Notes in Computer Science, vol. 8810, pp. 207–218. Springer (2014)
2. Ancona, D., Ferrando, A., Mascardi, V.: Comparing trace expressions and linear temporal logic for runtime verification. In: Theory and Practice of Formal Methods. Lecture Notes in Computer Science, vol. 9660, pp. 47–64. Springer (2016)
3. Ancona, D., Ferrando, A., Mascardi, V.: Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches. IJAOSE **6**(3/4), 402–442 (2018)
4. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4337, pp. 260–272. Springer (2006)
5. Brafman, R.I., Bar-Sinai, M., Ashkenazi, M.: Performance level profiles: A formal language for describing the expected performance of functional modules. In: IROS. pp. 1751–1756. IEEE (2016)
6. Chen, F., Roşu, G.: Towards monitoring-oriented programming: A paradigm combining specification and implementation. In: Workshop on Runtime Verification (RV'03). ENTCS, vol. 89(2), pp. 108 – 127 (2003)
7. Ferrando, A.: The early bird catches the worm: First verify, then monitor! Sci. Comput. Program. **172**, 160–179 (2019)
8. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Verifying and validating autonomous systems: Towards an integrated approach. In: RV. Lecture Notes in Computer Science, vol. 11237, pp. 263–281. Springer (2018)
9. Huang, J., Erdogan, C., Zhang, Y., Moore, B.M., Luo, Q., Sundaresan, A., Rosu, G.: ROSRV: runtime verification for robots. In: RV. Lecture Notes in Computer Science, vol. 8734, pp. 247–254. Springer (2014)
10. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
11. Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., Preoteasa, V.: Predictive runtime verification of timed properties. Journal of Systems and Software **132**, 353–365 (2017)
12. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
13. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)