ROMoSu: Flexible Runtime Monitoring Support for ROS-based Applications

Marco Stadler
Michael Vierhauser
LIT Secure and Correct Systems Lab
Johannes Kepler University Linz
{firstname}.{lastname}@jku.at

***Abstract*—ROS-based robotic applications are becoming increasingly common in various different application domains, performing diverse tasks. Examples include autonomous vehicles, small unmanned systems, as well as industrial applications of Cyber-Physical Production Systems. What all these systems have in common is their tight integration between hardware and software components, and close interactions with humans, e.g., on a shop floor, or autonomously driving robots as part of a warehouse system. This, in turn, requires monitoring the behavior of the system at runtime and ensuring that it behaves according to its specified requirements. However, establishing and maintaining runtime monitoring support is a non-trivial task, requiring significant up-front investment and extensive domain knowledge. To alleviate this problem, in this paper, we present ROMoSu, a flexible runtime monitoring framework for ROS-based systems that allows defining multiple scenarios, or application-specific configurations, taking into account different monitoring needs, and provides tool support for creating, maintaining, and managing configurations at runtime. As part of our evaluation, we have conducted experiments with three different use cases, of both physical and simulated applications. Results confirm that ROMoSu can be successfully used to create monitoring configurations with little effort, create efficient monitors and perform constraint checks based on the collected runtime data.**

***Index Terms*—ROS, Cyber-Physical Systems, Runtime Monitoring**

I. Introduction

ROS-based robotic applications have gained significant popularity in a wide variety of different domains, ranging from robotic applications to autonomous vehicles [1] and industrial applications of Cyber-Physical Production Systems [2]. What most Cyber-Physical Systems (CPS) have in common is their tight integration between hardware and software components, and their use in the presence of humans. For example, robotic arms operating as part of a shop floor automation system, or autonomously driving robots as part of a warehouse system [3]. This in turn requires various different operational aspects, such as safety, to be taken into consideration, to ensure that these systems behave as intended while performing their tasks. In this context, *Runtime Monitoring* has been introduced as a means for monitoring the behavior of a system at runtime, determining its correct and safe behavior, and detecting deviations from specified requirements by performing constraint checks on the collected runtime information.

The development of a software system that incorporates all these aspects, however, is not trivial, and establishing an effective runtime monitoring framework requires an adequate level of knowledge about the *System under Monitoring*'s (SuM) structure (sensors, actuators, software versions), its properties, and technologies. As part of a literature review [4], we analyzed over 350 runtime monitoring approaches used for different types of systems, and application domains and found that these approaches typically require a significant up-front investment to set up a monitoring framework, define constraints, and perform runtime checks. Therefore, developers are often forced to invest a significant effort, to adequately understand the SuM's specific characteristics and structure before they can establish solid runtime monitors. The additional effect of this high level of knowledge required is that novices, in particular, initially struggle with the development, and learning a new (domain-specific) technology to establish the monitoring can already become an overwhelming task [5].

Many modern robot applications rely on the *Robot Operating System* (ROS) [6] which provides a platform for a wide variety of different applications and systems and is one of the most popular frameworks for robotic applications, as well as autonomous systems. Monitoring applications built on top of ROS also require capabilities not only for collecting data, but for performing a subsequent analysis, and checking functional behavior, quality, or safety constraints.

To address these challenges, and provide support for runtime monitoring, in this paper we introduce **ROMoSu** (**RO**S **Mo**nitoring **Su**pport). ROMoSu enables users to easily create monitoring configurations and establish monitors for ROS applications in a straightforward fashion, independent of the SuM's structure and hardware components and with little prior knowledge of the system and its specific structure required. The work described in this paper significantly extends our previous work [7] where we collected challenges and derived capabilities for a monitoring framework. This includes a detailed architecture of our framework, a prototype implementation as well as an evaluation of ROMoSu.

The remainder of this paper is organized as follows: In Section II we present a motivating example and revisit the challenges associated with runtime monitoring. In Section III we present ROMoSu, our flexible runtime monitoring framework for ROS-based systems and describe its application

and prototype implementation. As part of our evaluation, in Section IV, we apply ROMoSu to three different types of ROS-based systems (two in the Gazebo simulation environment, and one using physical TurtleBot3 robots). Finally, we discuss related work in Section V and future work and conclusions in Section VI.

II. Challenges and Motivating Example

ROS-based systems have become the de facto standard for robotic applications and are used across various domains [8]. ROS is used in the context of *Cyber-Physical Production Systems* (CPPS) [9], [10], in the automotive domain [11], [12] or in agricultural robotics [13]. However, while ROS is frequently used in industry and research, as part of our previous work [7] with ROS-based and robotic applications, we have discovered the lack of generic runtime monitoring support and have identified six challenges (C1-C6) specifically pertaining to runtime monitoring. These challenges reflect concrete pain points when it comes to (1) establishing runtime monitoring support for ROS-based applications and (2) maintaining and updating the runtime monitors when new monitoring needs arise, or when the underlying SuM changes and evolves. Table I provides a brief summary of the challenges and resulting requirements for an efficient ROS runtime monitoring solution.

To exemplify these challenges and motivate the development of our ROMoSu framework, throughout the paper we use examples of Unmanned Aerial Vehicles (UAVs) frequently used in domains such as first response operations [14] or transportation and logistics [15]. UAVs employ ROS in a variety of scenarios [16]. For example, in a search-and-rescue operation, UAVs may be equipped with a gimbal and camera for searching for missing or drowning victims, while others may be equipped with a transportation device to drop flotation devices or medical supplies. Therefore, UAV systems involve a range of equipment, sensors, and actuators to perform their tasks as efficiently as possible that need to be monitored at runtime. Furthermore, software updates, or updates of the flight controller are deployed, resulting in structural changes in the system and leading to an evolving ecosystem. Keeping track of all these changes (especially in larger industrial contexts) is not feasible. Therefore, novices as well as domain experts profit from an initial overview of a system structure (cf. C1) to establish a co-evolving monitoring solution. Additionally, a monitoring solution specifically tailored to one type of UAV becomes cumbersome to maintain and variants need to be developed and deployed for different hardware versions. As a result, a monitoring solution must be able to handle a certain amount of flexibility (cf. C2).

UAVs provide numerous kinds of data, from initial startup checks (e.g., if a sufficient number of satellites are available), to distance measurements and rotor data. Brute-force monitoring of all the provided information requires significant resources (bandwidth, or computational) which may result in delays and even affect the SuM itself. Monitoring only a subset of properties (cf. C3) is crucial to ensure the correct behavior

	Chall.	Description & Monitoring Requirements
C1	*Initial Overview*	Provisioning of initial overview of the system structure; create an initial overview of the probably unknown system structure; support for automated collection of monitoring properties.
C2	*Flexbility*	Configure diverse monitoring needs; handle different (types of) data depending on their software version and hardware equipment (sensors, actuators, etc.).
C3	*Monitoring of Subsets*	Only a subset of system properties are likely to be monitored; not all data is of equal importance due to environmental circumstances, limited resources constrain runtime data collection.
C4	*Adaptive Collection & Analysis*	Properties need to be collected and analyzed with different frequencies; monitoring framework must adapt to the SuM's environment and users monitoring needs.
C5	*Constraint Checks*	Data must not only be collected but subsequently processed and analyzed, different domains/systems require diverse constraint checks that cover specified types of requirements.
C6	*Insights*	The outcome of the runtime monitoring data and services are accessible to the user; updates of the monitoring configuration when needed.

TABLE I: Overview of monitoring challenges for ROS applications and resulting Requirements [7].

of the system. Additionally, for example, in a scenario with multiple UAVs operating in a restricted area, the distance to the next UAV plays a pivotal role to avoid collisions. In a single UAV scenario, the importance might be significantly lower, motivating the need for adaptive data collection (cf. C4) as data should be collected and analyzed, e.g., in the form of constraint checks (cf. C5) in adequate frequencies with respect to their environmental circumstances. While executing missions with a UAV, the operators conducting the mission need insights (cf. C6) on the UAV data currently gathered and monitored and respond to possible problems as they arise.

III. The ROMoSu Framework

To address the aforementioned requirements, and to provide extended support for creating and maintaining monitoring configurations, ROMoSu provides support for the two main phases of runtime monitoring: first, the *Monitoring Configuration* phase (CT) where the user retrieves details about the system and creates one or more monitoring configurations, and second, at runtime while the system is performing its tasks, the *Monitoring Data Collection* phase (MT), data is collected and constraints are evaluated.

ROMoSu uses two main artifacts throughout these two phases. First, a `Mon-Config`, representing a monitoring configuration containing information about the publishing type and SuM, as well as a selection of topics (and sub-topics) that should be monitored, alongside frequencies specifying how often the collected data should be published by the framework. We distinguish between two publishing types: *complex* or *simple*. The former hereby preserves the original hierarchical topic structure provided by ROS, whereas the latter automatically flattens the nested topics and adds all of the monitoring data to the root topic when published.

Depending on the use case (and constraint checks that might be performed on the data) this gives the users the option to simplify their configurations if desired, or preserve the original ROS structure. Additionally, a SuM-Type specifies the types of systems that ought to be monitored with the framework, and where configurations exist. The mapping between a SuM-Type and Mon-Config is later on used to activate monitoring (i.e., instantiate a configuration at runtime) and further to make monitoring configurations reusable.

In the following, we provide a comprehensive overview of the architecture and component parts of ROMoSu and how both, the Mon-Configs and SuM-Types are created and used.

A. Framework Overview

An overview of the architecture is presented in Fig. 1. ROMoSu consists of four main parts: the Core components responsible for providing ROS data during the configuration and performing the actual runtime monitoring; the Admin UI used in the Configuration Phase for creating and maintaining Mon-Configs and SuM-Types; the Dashboard used during the actual Monitoring Data Collection Phase; and finally, external Services that allow to easily extend the framework with new capabilities and access the collected ROS data.

• **Framework Core:** The framework's core is divided into seven main components. The Connection Interface is used to communicate with components outside of the core and exchange data related to configuring the framework, Mon-Configs, and SuM-Types. The API Adapter hereby serves as the intermediary between the core and the connection interface, offering functions to serialize/deserialize data from other components. In addition, it serves as the connection point to the Config-Database, where all created configurations and SuM-Types are stored. Additionally, the ROS Adaptation Manager provides the business logic for starting and stopping ROS runtime monitoring based on the currently selected configurations, and triggers modules of the Runtime Data Broker to forward data with the specified frequency. To achieve this, the ROS Adaptation Manager caches the raw data provided by the ROS Connector in a Runtime Cache and retrieves the data again when needed. Data from the SuM is collected using the ROS Connector by subscribing to the desired ROS topics. and is subsequently distributed via the Runtime Data Broker.

• **Admin User Interface:** The Admin UI provides various UIs and editors to create and maintain artifacts including the SuM-Type-Editor. SuM-Types may be created, updated, and deleted with this component. The Configuration-Creator is responsible for allowing users to get an initial overview of the SuM's data structure and its supported monitoring topics (cf. C1 and C2). Furthermore, the users can specify the desired (sub-) topics (cf. C3), monitoring frequencies (cf. C4) and the SuM-Type it belongs to. The Config-Editor provides a user interface for managing monitoring configurations that have already been created, and updating and deleting topics and frequencies. Finally, the Monitoring Initializer sends activation triggers in the form of a selected monitoring configuration to start collecting monitoring data.

• **Dashboard:** The ROMoSu Dashboard UI consists of a Monitoring Supervisor and Data Explorer providing information about the status of active monitoring instances. Furthermore, the Data Explorer provides additional information about the actual monitoring data that is currently collected and distributed. This allows the user to confirm the correctness of the ongoing runtime monitoring behavior, and to gain insights about the system status (cf. C6).

• **Services:** ROMoSu follows a separation of concerns paradigm, allowing to connect external services via the Runtime Broker and tailoring the framework to specific needs, e.g. by attaching different types of constraint engines or databases for persisting runtime data. So far, we focus on two main services: Runtime Data Persistence, which is responsible for persisting the collected runtime data for a post-processing analysis, and Runtime Data Validation, which checks whether certain domain-dependent constraints have been violated during runtime (cf. C5). However, additional services such as data aggregation or processing services might be added to receive data via the Runtime Broker.

B. Applying ROMoSu

The process of configuration and subsequent monitoring and analysis follows three main process steps. First, a Mon-Config needs to be created using the Config-Editor (*Step 1*). Since ROS is topic-based, namespaces help distinguish different systems in a multi-system setting. For instance, in a setting in which three identical UAV models are operating simultaneously, their topics could use uav_n as a root topic to distinguish the identical published data, where n is the number of the respective UAV (here, 1-3). Even when just operating with a single system, it is considered to be best practice to assign namespaces to maintain a proper topic hierarchy [17].

The Config-Editor herby serves as the main user interface in this step, providing a list of possible ROS namespaces which corresponds to a single ROS system to be monitored. Next, the user is able to inspect the selected system structure. One can for example see at one glimpse, that a ROS Command Velocity Message consists of two 3D vectors (x, y and z), one for the linear and one for the angular velocity (cf Fig. 2). Besides the name of the sub-topics, the respective data types are also provided. This allows making an informed decision on what configuration parameters should be monitored by (sub-) topics and respective frequencies.

Once a configuration has been created, the second step uses the Monitoring Initializer to activate the actual runtime monitoring, data collection, and analysis (*Step 2*). Again, a namespace needs to be selected which should be monitored, and a dedicated SuM-Type needs to be selected which in turn lets the user select one of the previously created configurations for this particular type. This makes it possible to specify multiple different configurations (e.g., for different execution scenarios) for a SuM-Type in advance and then initialize the

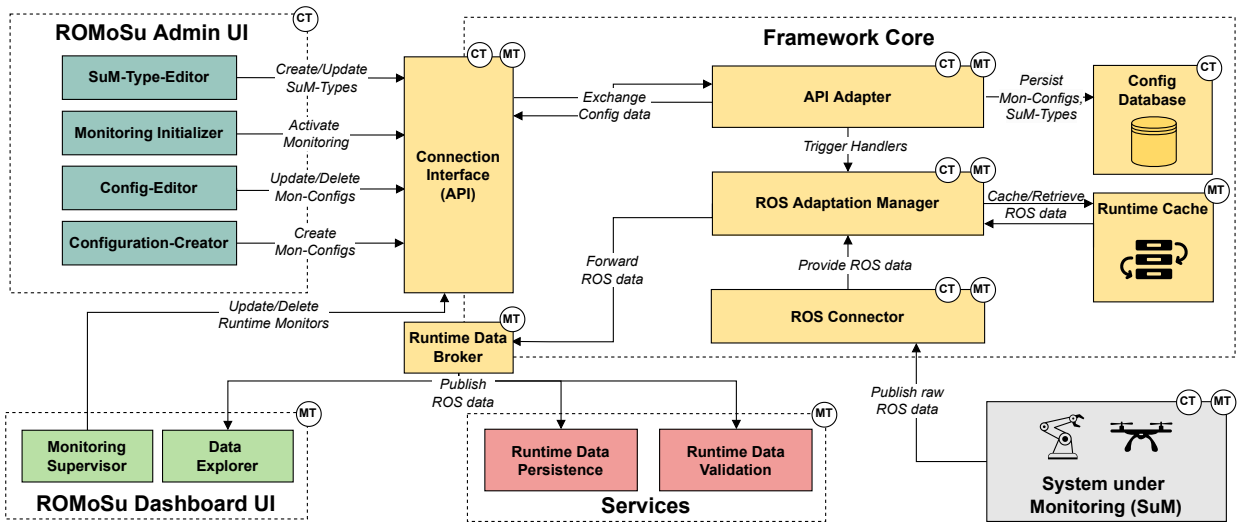


Fig. 1: Abstract overview of ROMoSu's component structure and the four main components. (The CT/MT circles refer to the phases of ROMoSu, when the respective component is used).

desired configuration on demand (with specific topics and frequencies as part of the configuration).

Finally, to make use of the collected data, diverse services can be attached and used (*Step 3*). Note, in contrast to *Step 1* and *Step 2* these steps do not need to be executed in the presented order. As part of our current implementation (cf. Section IV) we provide 2 specific examples (data persistence and runtime constraint checking) of services using the runtime data provided by the `Runtime Data Broker`. By using a `Runtime Data Validation` component, constraint checks can be performed at runtime. Finally, by deploying a `Runtime Data Persistence`, runtime data coming from the Runtime Data Broker can be persisted for a post-mortem analysis. We provide further details about currently available services in Section IV as part of our prototype implementation and evaluation. Additionally, users can interact with the `Dashboard UI` to supervise ongoing monitoring instances and to explore data collected. This, for example, includes the possibility to inspect metadata (selected namespaces/ROS systems, time when monitoring was initiated, assigned SuM-Types, etc.) and also terminate an active monitoring configuration to halt further data collection. Furthermore, to perform manual data inspection and validation, it is possible to explore recent data collected by ROMoSu with the Data Explorer.

C. Prototype Implementation

Based on the described architecture, we have implemented a prototype of our framework which we subsequently use for our experimental evaluation (Section IV). The technology stack of the core and service components comprises an Angular frontend, Django backend, and SQLite and InfluxDB databases to persist the Mon-Configs, SuM-Types, and runtime data.

• **User Interface:** All UI components (Dashboard and Admin UI) are realized using individual views part of an Angular application. For instance, the Config-Editor (cf. Fig. 2) is designed as a four-steps wizard, enabling the user to easily create new configurations via the user interface. The other components follow the same principle providing capabilities for managing configurations and initiating/stopping data collection.

• **Framework Core:** The frontend applications use a RESTful API provided by the Django framework [18] (Connection Interface) to communicate with the server and the ROS Adaptation Manager that handles runtime data. When a new configuration is activated, the Adaptation Manager dynamically instantiates a multi-threaded environment (one per topic) to handle the individual monitoring frequencies. The connection to the ROS application (via the ROS Connector) is implemented using `roslibpy`, an open-source Python-based middleware. We leverage its functions for topic subscriptions and gathering meta-information about the ROS systems currently connected. The Runtime Data Broker is implemented using a Mosquitto MQTT broker that, similar to ROS itself, uses a publish-subscribe-based protocol, commonly used for high-performance message transmission in the IoT domain. Data is published in a JSON format and depends on the selected publish type, either on the root topic or nested subtopics.

• **Services:** As proof of concept, we implemented a persistence service using InfluxDB, a time series database running in a separate Docker container receiving data via the Runtime Data Broker. For runtime validation, we use Esper [19], a Complex Event Processing (CEP) engine. This allows us to not only check static property values (for example, if a value is below or above a certain threshold), but also temporal checks, for example, if certain conditions are violated over time (e.g., did the average value of the last ten seconds exceed a threshold). We provide further examples of constraints in our evaluation (cf. Section IV). Same as the persistence service, the validation service operates as a completely independent component implemented in Java, receiving data via the Runtime Data Broker.

IV. Experimental Evaluation

To demonstrate ROMoSu's applicability to real-world applications, and its capabilities for defining monitoring configurations and performing runtime monitoring, we selected three use cases of ROS-based applications. The focus of the evaluation was to first to demonstrate general *feasibility* and applicability, and second to assess the *performance* of our approach when performing data collection and constraint checks. We, therefore, explore the following research questions:

RQ1: Can ROMoSu be used to create monitoring configurations for diverse systems and perform subsequent analysis tasks, and what is the effort required?

RQ2: Can ROMoSu be used efficiently to collect runtime data and perform subsequent analysis tasks?

A. Evaluation Setup

With the first research question, the goal was to evaluate the general feasibility of our approach, and applicability of ROMoSu to different ROS-based systems. The first system (*GTB*) is a simulation of TurtleBot3 [20] robots using the ROS standard simulation software *Gazebo* [21]. A TurtleBot is a small ROS-based mobile robot used frequently for prototyping in research and education. This first scenario used Gazebo's multi-robot simulation to spawn three TurtleBot models. The data provided by the simulated robots (such as odometry, system status, etc.) are then used as input for the evaluation, creation of monitoring configuration, and subsequent analysis.

The second scenario (*HTB*) used ROMoSu in a similar example, however, instead of a simulation environment, utilizing physical TurtleBot3 robots to demonstrate that the framework can handle both simulated and "real" robotic applications. In contrast to the simulation, this use case provided a wider variety of ROS topics including information about the used firmware versions in the hardware, battery status, and diagnostics data from sensors and actuators.

Finally, for the last case (*GMX*) we selected an entirely different ROS application, a Gazebo simulation of the OpenMANIPULATOR-X [22], an open software, open hardware, and OpenCR (embedded board) robotics gripper arm which is also frequently used in research context [23].

To answer RQ1, we used our prototype implementation and generated five Mon-Configs for each scenario with significantly different complexity, meaning the number of topics selected, the frequencies specified and the publish type selected. In a second step (RQ2), we wanted to ensure that ROMoSu can efficiently handle monitoring data and operate under realistic conditions in both simulated and physical environments. For this purpose, we focused on three aspects: (1) comparison of the potential reduction in event data; (2) receive transmit time (RTT) and effective processing time (EPT); and (3) suitability of the constraint validation service.

RTT measures the time from the arrival of a ROS message until it is forwarded by the Runtime Data Broker, i.e., the total time the data rests in the system. However, it is important to note that this metric is also dependent on the frequency configuration and the native ROS internal publishing interval of the respective topics, but still provides an indication of whether significant delays occur within the framework. Therefore, we also measured the effective processing time (EPT), which corresponds to the time ROMoSu takes to save a newly arrived topic from ROS in the cache plus the time it takes to again retrieve the event from cache and forward it to the MQTT Broker. Finally, to assess the Runtime Data Validation service, we seeded random "errors" into the published data stream to check whether constraint checks are correctly triggered and violations are detected by the constraint engine. For each use case, we selected three constraints based on the available topics provided by the simulation or hardware component.

CST	Description	Type	NCST [#]
GTB1	*Movement Speed Limit:* For safety reasons, the bot must not move faster than 0.35 m/s.	S	13
GTB2	*Obstacle Avoidance:* The bot must maintain a minimum distance of 35 cm to an object, as detected by the LiDAR unit.	S	34
GTB3	*Navigation Error:* To maintain accuracy during navigation, an average of 30 of the 360 LaserScan angle bins must detect a distance value over 10 s.	T	54
HTB1	*Speed Limit:* To maintain accuracy during navigation, the bot must not move faster than 0.35 m/s.	S	30
HTB2	*Obstacle Avoidance:* The bot must maintain a minimum distance of 35 cm from an object, as detected by the LiDAR unit.	S	69
HTB3	*Speed Reduction:* When below 25 % battery level, the avg speed must not exceed 0.1 m/s.	T	21
GMX1	*Joint Effort Limit:* The consumption of each DYNAMIXEL joint effort must not exceed 5.0 Nm.	S	45
GMX2	*Gripper Range Limit:* The gripper must not operate below the robot's root surface ($z > 0.0$).	S	31
GMX3	*Gripper Opening:* To ensure accuracy, after moving, the gripper must not open for 2 s.	T	27

TABLE II: Constraints for static property value checks (S) and temporal checks (T) used in the evaluation. NCST describes the mean violations reported in our seeded runs.

For each measurement, for each use case, we performed three evaluation runs (i.e., 45 runs in total) with each lasting three minutes and report mean values.

B. Results

For each of the three use cases, we performed the steps described in Section III-B and then performed our evaluation runs.

RQ1: As a precursor, we connected the ROS systems to the roscore instance and started the respective ROS application. Using ROMoSu for creating two SuM-Types (one for GMX and one for HTB/GTB) was a straightforward task, only requiring the respective user interface (SuM-Type Editor). The SuM-Type created for the GTB could then also be reused for the HTB as all topics provided by the GTB are also covered by the HTB. For creating the monitoring configurations, we again used the provided user interface. ROMoSu provides high flexibility with regard to different SuM, topic structures, and topics. In the GTB use case, three different namespaces (one per TurtleBot) were automatically created. In contrast, for the

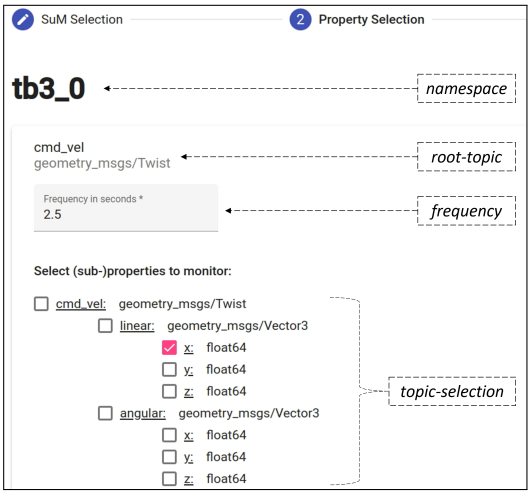


Fig. 2: ROMoSu UI for creating `Mon-Configs`

GMX use case, only a single namespace for the device was specified, with a total number of 11 unique topics.

Additionally, frequencies were specified for each root topic, and data was published later on accordingly. The overall specification process of a single Mon-Config took less than 3 minutes (performed by one author of the paper). However, this is highly dependent on the time spent on analyzing the structure and deciding which data to be analyzed). Once the configurations were created, we used the Monitoring Initializer to activate the defined Monitoring-Configs. The activation wizard only required the selection of a namespace, SuM-Type, and Mon-Config to activate the runtime monitoring and establish monitors. The total time effort for creating a Mon-Config and collecting runtime data with the Mon-Config took less than 10 minutes.

Upon activation, runtime data was collected and stored in the InfluxDB with the dashboard providing information about the ongoing runtime collection. In order to validate that the requested Mon-Config was properly configured and provided the specified data, the data explorer was used to inspect the runtime data. An overview of the checked constraints in our evaluation can be found in Table II. We distinguish constraints into two types: Static value checks verify that certain values should not exceed/fall below a certain threshold. They are time-independent in the sense that these checks are performed on every single event processed by the engine. Temporal checks on the other hand check thresholds over a specified time. These constraints tend to be more complex as they require multiple events to be checked. For example, Turetle-Bots use nearby (detected) objects to navigate. Inaccurate positioning and navigation may occur if the distance exceeds the LaserScan's sensor sensitivity. Determining the average number of LaserScan bins over a 10-second period, GTB3 is used to alert the operator that navigation may not function as intended if the constraint is violated.

In summary, we were able to create a total number of 15 configurations (five per use case) with different topics to be monitored and different frequencies with which the data is collected and forwarded by our framework. ROMoSu itself supplied all the information required to set up the runtime monitoring. To choose which topics are offered and may be monitored, no other tools (such as the ROS rqt Topic Monitor) were required. Answering *RQ1*, we can conclude that ROMoSu can be used to easily create monitoring configurations for different kinds of systems, topics, frequencies, and monitoring needs. When needed, configurations can be easily selected and activated and the desired data is automatically forwarded to the Runtime Data Broker, readily available for the external services to be used.

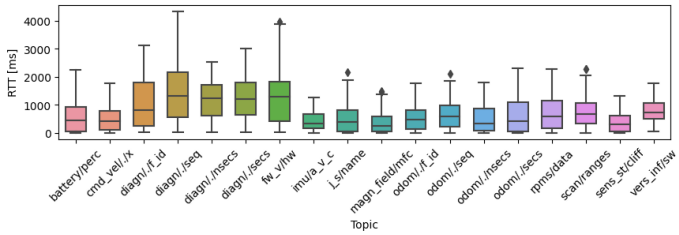
RQ2: An overview of the potential event reduction results can be found in Table III. For the Brute-Force Monitoring runs we selected all topics which were provided by the ROS systems (e.g., 13 topics for HTB) with their native frequencies. For the selective monitoring, we specified a Mon-Config which is needed to perform the constraint checks and frequencies based on the assumption that non-safety critical constraints, for example, do not need to be checked 50 times a second. For HTB, three topics are required to support the constraint checks and as a consequence, only three of the 13 topics have been selected for selective monitoring. In each of the three use cases, ROMoSu was able to significantly reduce the number of messages. The highest decrease has been recorded for GTB – 4,015 messages less than with the brute-force monitoring which is equivalent to a reduction of 95.48 %.

Performance-related results are shown in Table III. Empty topics that do not contain data were omitted from the examination to prevent distortion of the measurements (4 topics of GMX were removed). Furthermore, to analyze the influence of data types on the performance metrics, we monitored multiple subtopics with different data types of the same root topic (5 additional topics by HTB). While the RTT values for GTB and GMX do not substantially differ from each other (difference of 48 ms), the RTT value of HTB is significantly higher (907 ms). This difference can be primarily attributed to the ROS-specific publishing times, as the timeliness of the data is dependent on this publishing interval. The distribution of HTB RTT values are displayed in Fig. 3a. The median values of the `firmware_version` is almost 1 second higher than the `odom` topics. This is due to the fact that per default the maximum publishing rate of the `firmware_version` topic is 1 Hz, while the `odom` topic is published at a maximum default rate of 30 Hz. These rates are further (negatively) influenced by other factors, such as battery health, CPU usage and bandwidth. We also found that data types seem not to influence the processing time. The results for the EPT show that the influence of ROMoSu on the delay is not significant since the total average EPT of ROMoSu is 0.27 ms considered noticeably low for our given application use cases.

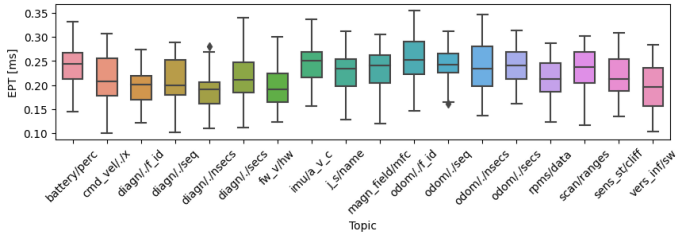
Integrating the published data into the stand-alone Java-based constraint engine was simple. We found that the constraint engine detected constraint violations as expected. (cf. Table II). The NCST values in the table differ in their amount for two reasons, first, the violations depend not only on a simple randomness function but also on the publishing frequency specified in the Mon-Config. While GTB2 and

System	Brute-Force		Selective		Efficiency	
	Topics [#]	*Msgs [#]*	*Topics [#]*	*Msgs [#]*	*RTT [ms]*	*EPT [ms]*
GTB	5	4,205	2	190	395	0.31
HTB	13	1,462	3	175	907	0.22
GMX	15	1,358	5	202	443	0.29

TABLE III: Evaluation results of ROMoSu (mean values)



(a) RTT data visualization



(b) EPT data visualization

Fig. 3: RTT and EPT boxplots of HTB

HTB1 had an equal publishing frequency and a 25 % chance of publishing error data, the frequency of GTB1 was lower, resulting in fewer constraint violations. In conclusion, the runtime data validation behaved as intended and, therefore, ROMoSu is capable of supplying ROS runtime data to constraint engines and performing runtime checks.

To answer *RQ2*, ROMoSu can be used for an efficient collection of ROS runtime data. The discussed measurements yielded good results, with the EPT in particular being deemed effective. A later analysis employing a constraint engine demonstrated the viability of integrating ROMoSu data with external services.

C. Discussion

One aspect that turned out to be highly beneficial was the ability to create both complex, as well as simple MonConfigs. Depending on the monitoring scope (i.e., the number of selected topics to be monitored, either the former or the latter can be selected by the user). When only few topics are monitored, combining them in a single JSON element greatly reduces the complexity and effort for writing subsequent CEP Esper constraints, whereas for a more exhaustive monitoring configuration preserving hierarchical structure may increase complexity, however, results in a more "logical" structuring of the monitoring data. Since ROMoSu operates independently of the underlying system, configurations can also be changed and updated independently even switching configurations while running the Gazebo simulations, and if so desired even monitoring the Gazebo simulation environment itself, as it also publishes its metadata via ROS topics.

D. Threats to Validity

Concerning generalizability of our results and findings, we have applied our framework in the context of three different use cases. While the first and second use case both use TurtleBot3 they differ in terms of available topics and collected data. Furthermore, for the third use case, we have selected a new system with an entirely different topic structure. To further assess the generalizability and broader applicability of our ROMoSu framework, additional case studies are required.

The initial implementation and evaluation runs use a limited number of topics and constraints, however, we have not observed any degradation over time when executing scenarios for a longer amount of time. Furthermore, we are using proven, off-the-shelf components for message transportation (MQTT) and constraint checking (CEP) and we are therefore confident that ROMoSu can handle a larger number of topics and amount of events. A more exhaustive evaluation is required to further evaluate the scalability with regard to messages and constraint checks. So far, we have used ROMoSu only internally, for creating monitoring configurations. However, we found that for all use cases, we were able to represent and monitor all specified topics and constraints. To further asses the usability and usefulness of our framework, we are planning on conducting a user study specifically focusing on the various steps in the defined process.

V. Related Work

A wide variety of different runtime monitoring solutions have been proposed, providing various types of monitors, constraint checks, and visualization options [24]. Particularly, in the domain of ROS-based applications, several approaches provide support for monitoring and verification of ROS topics and different types of constraints. For example, Ferrando et al. [25] present ROSMonitoring, a framework for runtime verification of ROS-based robotic applications. Similar to our approach, their work focuses on automated verification of the communication between ROS nodes by monitoring topics and checking against formal properties expressed. Huang et al. [26] present ROSRV, a runtime verification framework focusing on monitoring safety and security properties for robot applications.

Similar work by Adam et al. [27] and Shivakumar et al. [28] uses Domain-Specific Languages (DSL) to declaratively define safety-related rules and run-time assurance safety guarantees. While such a DSL could serve as an extension for our MonConfigs to declaratively specify configurations, in our work with ROMoSu, we focus on the monitoring level, providing capabilities for adaptive monitoring (frequency) and additional external services that can perform diverse constraint checks.

Focusing on QoS requirements, Parra et al. [29] present a DSL for defining communication QoS profiles for ROS 2 applications. While their approach also provides support for monitoring topics, the main purpose is to check and ensure certain quality metrics for topics, rather than an overall monitoring framework. Witte and Tichy [30] present a communication pattern and library for provenance tracking of

ROS 2 messages, with the main purpose of linking values to their origin across multiple nodes. Mengthi et al. [31] present PsALM, a toolchain supporting the development of dependable robotic missions. While their approach provides formal specifications and checks for robotic missions using LTL and CTL temporal logic, their focus is not on monitoring, but rather on overall mission specification and execution.

VI. Conclusion

In this paper, we have presented ROMoSu, a flexible monitoring framework for ROS-based applications. ROMoSu consists of UIs for inspecting the SuM, creating monitoring configurations, and exploring runtime data; a core responsible for configuration management and data collection; as well as additional services. As part of our prototype implementation, we have implemented services for data persistence using a time series database and complex event processing for constraint checking. Our evaluation – using three different, both simulated and real physical application use cases – has demonstrated that ROMoSu is capable of capturing relevant system properties and establishing efficient runtime monitoring support. As part of our future work, we are working on extending our approach with additional services, and more diverse constraint checks. Additionally, we intend to provide adaptive configuration switching not just during the configuration phase, but also at runtime (e.g., configuration switch for a UAV during flight vs. on the ground). We are further committed to making our approach publicly available as an open-source application.

References

[1] G. C. Medina, J. Guiochet, C. Lesire, and A. Manecy, "A skill fault model for autonomous systems," in *Proc. of the 4th IEEE/ACM Int'l Workshop on Robotics Software Engineering*. IEEE, 2022, pp. 55–62.

[2] C. Mayr-Dorn, M. Winterer, C. Salomon, D. Hohensinger, and R. Ramler, "Considerations for using Block-Based Languages for Industrial Robot Programming - a Case Study," in *Proc. of the 3rd IEEE/ACM Int'l Workshop on Robotics Software Engineering*, Jun. 2021, pp. 5–12.

[3] C. V. Lozano and K. K. Vijayan, "Literature review on cyber physical systems design," *Procedia Manufacturing*, vol. 45, pp. 295–300, 2020.

[4] M. Vierhauser, R. Rabiser, and P. Grünbacher, "Requirements monitoring frameworks: A systematic review," *Information and Software Technology*, vol. 80, pp. 89–109, 2016.

[5] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: a perspective from the service robotics domain," in *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020, pp. 593–604.

[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "ROS: an open-source Robot Operating System," in *Proc. of the ICRA Workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[7] M. Stadler, M. Vierhauser, and J. Cleland-Huang, "Towards flexible runtime monitoring support for ros-based applications," in *Proc. of the 4th IEEE/ACM Int'l Workshop on Robotics Software Engineering*, 2022, pp. 43–46.

[8] M. Albonico, M. Đorđević, E. Hamer, and I. Malavolta, "Software engineering research on the Robot Operating System: A systematic mapping study," *Journal of Systems and Software*, vol. 197, p. 111574, Mar. 2023.

[9] C. A. Garcia, W. Montalvo-Lopez, and M. V. Garcia, "Human-Robot Collaboration Based on Cyber-Physical Production System and MQTT," *Procedia Manufacturing*, vol. 42, pp. 315–321, Jan. 2020.

[10] A. Hennecke and M. Ruskowski, "Design of a flexible robot cell demonstrator based on CPPS concepts and technologies," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, May 2018, pp. 534–539.

[11] A.-M. Hellmund, S. Wirges, O. S. Tas, C. Bandera, and N. O. Salscheider, "Robot operating system: A modular software framework for automated driving," in *Proc. of the 19th IEEE Int'l Conference on Intelligent Transportation Systems*, 2016, pp. 1564–1570.

[12] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A Self-Driving Car Architecture in ROS2," in *Proc. of the 2020 Int'l SAUPEC/RobMech/PRASA Conference*, 2020, pp. 1–6.

[13] N. Tsolakis, D. Bechtsis, and D. Bochtis, "AgROS: A Robot Operating System Based Emulation Tool for Agricultural Robotics," *Agronomy*, vol. 9, no. 7, p. 403, Jul. 2019, number: 7 Publisher: Multidisciplinary Digital Publishing Institute.

[14] M. N. A. Islam, M. T. Chowdhury, A. Agrawal, M. Murphy, R. Mehta, D. Kudriavtseva, J. Cleland-Huang, M. Vierhauser, and M. Chechik, "Configuring mission-specific behavior in a product line of collaborating Small Unmanned Aerial Systems," *Journal of Systems and Software*, vol. 197, p. 111543, Mar. 2023.

[15] M. Sajid, H. Mittal, S. Pare, and M. Prasad, "Routing and scheduling optimization for UAV assisted delivery system: A hybrid approach," *Applied Soft Computing*, vol. 126, p. 109225, Sep. 2022.

[16] K. Aloui, M. Hammadi, A. Guizani, M. Haddar, and T. Soriano, "A new SysML Model for UAV Swarm Modeling: UavSwarmML," in *Proc. of the 2022 IEEE Int'l Systems Conference*, Apr. 2022, pp. 1–8.

[17] ROS Wiki, "ROS-Wiki: Graph Resource Names," 2022, [Last accessed 01-01-2023]. [Online]. Available: http://wiki.ros.org/Names

[18] Django, "Python web framework," [Last accessed 01-01-2023]. [Online]. Available: https://www.djangoproject.com

[19] EsperTech Inc., "Esper," https://www.espertech.com/esper, 2023, [Last accessed 01-01-2023].

[20] Open Source Robotics Foundation, "TurtleBot," [Last accessed 01-01-2023]. [Online]. Available: https://www.turtlebot.com

[21] Open Robotics, "Gazebo," https://gazebosim.org/home, 2023, [Last accessed 01-01-2023].

[22] ROBOTIS, "E-Manual OpenMANIPULATOR-X," https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview, 2023, [Last accessed 01-01-2023].

[23] D. H. Kwon and R. Gebhardt, "An Affordable, Accessible Human Motion Controlled Interactive Robot and Simulation through ROS and Azure Kinect," in *Proc. of the 2021 ACM/IEEE Int'l Conference on Human-Robot Interaction*. ACM, 2021, pp. 649–650.

[24] R. Rabiser, K. Schmid, H. Eichelberger, M. Vierhauser, S. Guinea, and P. Grünbacher, "A domain analysis of resource and requirements monitoring: Towards a comprehensive model of the software monitoring domain," *Information and Software Technology*, vol. 111, pp. 86–109, 2019.

[25] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "Rosmonitoring: a runtime verification framework for ros," in *Proc. of the Annual Conference Towards Autonomous Robotic Systems*. Springer, 2020, pp. 387–399.

[26] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, "Rosrv: Runtime verification for robots," in *Proc. of the Int'l Conference on Runtime Verification*. Springer, 2014, pp. 247–254.

[27] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz, "Towards rule-based dynamic safety monitoring for mobile robots," in *Proc. of the Int'l Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 207–218.

[28] S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia, "Soter on ros: a run-time assurance framework on the robot operating system," in *Proc. of the Int'l Conference on Runtime Verification*. Springer, 2020, pp. 184–194.

[29] S. Parra, S. Schneider, and N. Hochgeschwender, "Specifying qos requirements and capabilities for component-based robot software," in *Proc. of the 3rd Int'l Workshop on Robotics Software Engineering*. IEEE, 2021, pp. 29–36.

[30] T. Witte and M. Tichy, "Inferred interactive controls through provenance tracking of ros message data," in *Proc. of the 3rd Int'l Workshop on Robotics Software Engineering*. IEEE, 2021, pp. 67–74.

[31] C. Menghi, C. Tsigkanos, T. Berger, and P. Pelliccione, "Psalm: Specification of dependable robotic missions," in *Proc. of the 2019 IEEE/ACM 41st Int'l Conference on Software Engineering: Companion Proceedings*. IEEE, 2019, pp. 99–102.