Towards flexible Runtime Monitoring Support for ROS-based Applications

Marco Stadler
marco.stadler@jku.at
Dept. of Business Informatics – Software Engineering
Johannes Kepler University Linz
Linz, Austria

Michael Vierhauser
michael.vierhauser@jku.at
LIT Secure and Correct Systems Lab
Johannes Kepler University Linz
Linz, Austria

Jane Cleland-Huang
janeclelandhuang@nd.edu
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, United States

ABSTRACT

Robotic systems are becoming common in different domains and for various purposes, such as unmanned aerial vehicles performing search and rescue operations, or robots operating in manufacturing plants. Such systems are characterized by close interactions, or even collaborations, between hardware and machinery on the one hand, and humans on the other. Furthermore, as Cyber-Physical Systems (CPS) in general and robotic applications in particular typically operate in an emergent environment, unanticipated events may occur during their operation, making the need for runtime monitoring support a crucial yet often time-consuming task. Runtime monitoring typically requires establishing support for collecting data, aggregating and transporting the data to a monitoring framework for persistence and further processing, and finally, performing checks of functional and non-functional properties. In this paper, we present our initial efforts towards a flexible monitoring framework for ROS-based systems. We report on challenges for establishing runtime monitoring support and present our preliminary architecture that aims to significantly reduce the setup and maintenance effort when creating monitors and establishing constraint checks.

CCS CONCEPTS

• **Computer systems organization** → *Robotics.*

KEYWORDS

ROS, Cyber-Physical Systems, Runtime Monitoring

ACM Reference Format:
Marco Stadler, Michael Vierhauser, and Jane Cleland-Huang. 2022. Towards flexible Runtime Monitoring Support for ROS-based Applications. In *4th International Workshop on Robotics Software Engineering (RoSE'22), May 9, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3526071.3527515

1 INTRODUCTION

Robotic systems are currently used across different domains, for various purposes and diverse tasks. Examples include unmanned aerial vehicles, e.g., performing search and rescue operations [1], and industrial applications of Cyber-Physical Production Systems [10]. What most Cyber-Physical Systems (CPS) have in common is a close interaction, or even collaboration between hardware and machinery on the one hand, and humans on the other. Furthermore, as CPS typically operate in an emergent environment, unanticipated events may occur during operation. This, in turn, requires to monitoring the behavior of the robot at runtime, collecting runtime information and checking safety properties. Regarding safety, various solutions have been proposed and approaches have been developed to assure the safe behavior of such systems and to perform mitigating actions if critical safety constraints are violated [2, 9, 16].

However, regardless of the application domain, type of system, or monitoring objective, most approaches so far have been developed with a specific purpose in mind, typically supporting only a narrow set of features or type of system. In a systematic literature review [18], we analyzed over 350 runtime monitoring approaches used for different types of systems, constraints, and technologies, and found that these approaches all require a significant up-front investment in order to set up a monitoring framework, define constraints, and perform checks on the collected runtime data.

Many modern robot applications rely on the Robot Operating System (ROS) which provides a platform for a wide variety of different applications and systems. Applications built on top of ROS also require capabilities not only for collecting data, but for performing subsequent analysis, and checking functional behavior, quality or safety constraints. In order to avoid reinventing the wheel, i.e. implementing a new monitoring framework every time a new ROS-based application is built, in this paper we describe our initial concepts and efforts towards a flexible, yet easy to use runtime monitoring framework for ROS-based systems. Model-driven Engineering (MDE), combined with models@runtime [3] has proven valuable for modeling design-time aspects of the system, managing and maintaining runtime data, and also for addressing evolution and maintenance. Therefore, based on previous initial work on MD-monitoring [17], in this paper, we identify challenges for providing effective monitoring support for ROS-based systems and subsequently discuss requirements of a monitoring framework.

The remainder of this paper is organized as follows: Section 2 discusses monitoring challenges and Section 3 introduces the concept of a flexible runtime monitoring framework for ROS-based systems. Finally, we discuss our planned research in Section 4 and conclude the paper in Section 5.

RoSE'22, May 9, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9317-1/22/05…$15.00
https://doi.org/10.1145/3526071.3527515

2 MONITORING CHALLENGES

CPS are used in a variety of application areas, including industry applications such as factory automation or transportation. A shared understanding of the perhaps (partially) unknown structure of the system, and the data provided by the different parts of the system, is crucial to enable meaningful runtime monitoring and to allocate the collected data and the resulting violations and warnings to the respective part of the system.

Thus, one major challenge resides in the **provisioning of an initial overview of the system structure (C1)**. The user makes decisions about the monitoring characteristics, and ultimately the monitoring configuration (*what* needs to be monitored, *when*, and *how* frequently). As CPS consist of diverse components such as robotic arms, automated storage, and autonomous robots in an industrial manufacturing environment, one needs to know what kind of data is available and which parts are in fact relevant for subsequent runtime checks [7]. Depending on the robot's sensors, this could, for example, include data about the battery status, velocity, or data from a LiDAR unit (e.g., distance to the nearest obstacle).

Furthermore, different systems provide different (types) of data, depending on the version of the system, and individual machines may be equipped with unique sensors and actuators depending on the tasks they are performing. A one-size-fits-all monitoring solution might be capable of collecting general properties but does not align well with the individual needs for certain parts of the CPS. A second monitoring challenge, therefore, stems from this **diversity and need to individually configure monitoring needs (C2)** for different parts of a CPS. Without a proper and easily adaptable monitoring framework, important data might be missed, or a significant amount of effort is required to customize and update the monitoring framework once the System under Monitoring (SuM) changes.

While some systems provide documentation of its components (e.g., using UML) and hence its monitorable properties, **only a subset of these properties likely need to be monitored on a continuous basis (C3)**. Furthermore, depending on the task, or environmental factors, data might need to be collected and analyzed more frequently to ensure safe behavior, or less frequently to preserve battery power. Therefore, adaptive monitoring [4, 5] plays an important role, meaning that the monitoring infrastructure itself needs to adapt depending on environmental conditions and/or the state of the system. For instance, if a robot is in an environment where it travels along a predefined path with fixed stops, **data needs to be collected and analyzed differently (C4)** (i.e., less frequently) compared to when it is operating in a dynamic environment where the robot can move freely. In the dynamic environment, the robot may encounter another robot during navigation, requiring data (like distance to nearest obstacle) to be updated more frequently to avoid collisions. In addition to environmental conditions, the internal state of the robot might also affect monitoring. For example, when a robot's battery is running low, certain non-critical properties might be collected less frequently in order to preserve battery and prolong the current operation [8]. This has a trickle-down impact by reducing subsequent processing and analysis of the data. While in some cases local monitoring on the robot itself is sufficient, in other cases central monitoring is more appropriate. For example, if multiple robots within a delineated area are initially activated and do not move, the current location does not need to be transmitted to a central unit. However, if several robots perform different tasks and move simultaneously within the area, the location of the robots is important and should be transmitted to a central unit.

Typically in a monitoring framework, data is not only collected but then subsequently processed and analyzed. Depending on the type of data, and the application scenario, **diverse types of constraints need to be defined and checked on the data (C5)**. For example, safety zones must be maintained to ensure safety while humans and machines are working simultaneously. Besides runtime evaluation of constraints, another use case is post-mortem analysis of data stored over time, making it necessary to also persist the collected data for later use [6, 13].

Finally, it is important to **make the outcome of the runtime monitoring data and services accessible to the user (C6)**. This includes providing adequate information about the output of the services including violations of constraints, collected runtime data, and the persisted data. This way, the user can make sure that everything is working properly according to the current state of the robot and its operating environment. Simultaneously, this initiates a re-evaluation process of the defined user configuration. For example, if a constraint is continuously violated during monitoring, the user is guided to initiate appropriate responses, including verifying that the physical SuM is functioning properly or that a redefinition of constraints is required.

3 A FLEXIBLE RUNTIME MONITORING FRAMEWORK FOR ROS-BASED CPS

To address the previously mentioned challenges, we present our initial version of a flexible framework for runtime monitoring support of ROS-based applications, as shown in Fig. 1. Our framework consists of six main components: a *System Modeler* for creating a monitoring configuration (definition and selection of monitoring properties of the SuM); a *Monitor Adaptation Manager* for defining and executing adaptation of the monitors; the *Runtime Model* for collecting and distributing monitoring information; *Constraint Engine(s)* for performing runtime checks; a *Data Persistence* component for storing collected monitoring data; and a *Runtime Dashboard* providing evaluation results and insights to the user.

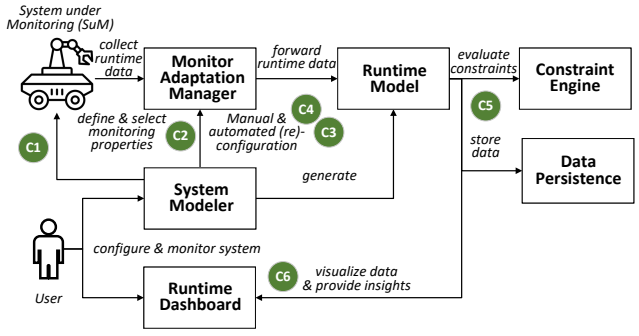

Figure 1: High-Level overview of our monitoring approach

3.1 Overview

• **System Modeler:** Establishing a monitoring infrastructure can be accomplished in a top-down fashion, where all systems, elements, and properties are defined beforehand, and then the respective data from various components of the system is collected. Alternatively, a bottom-up approach can be used in which the actual data serves as a starting point, and the definition of the monitored components is built around it. As ROS provides a topic-based subscription service, information about the data, and rudimentary information about the components – encoded in the topic structure – can be retrieved at runtime and leveraged to incrementally build a model of the SuM. As part of our approach, we propose a combination of both the top-down and bottom-up approach, where the System Modeler component connects to the running system and provides the user with information about currently active topics and data that is sent. The user can then visually access and explore the properties of the SuM in a user interface (C1). Based on this information, the user can select/update important properties that are of interest for monitoring and link them to the respective ROS topics. Additional properties can be added, shaping the hierarchical structure and dependencies of the system. In turn, this structure is then used for activating or deactivating properties in the running system, and stored as a monitoring configuration (C2). Additionally, in order to be able to react to different states or environmental conditions, state transition diagrams are used to define different states that require different properties to be monitored (C3). The states are then associated with the properties defined in the monitoring configuration and used to define which property needs to be monitored in which state. Furthermore, for each state, the frequency of which data is collected can be specified on a per-property basis (C4).

• **Monitor Adaptation Manager:** Once the structure of the system and monitoring states are defined, the Monitor Adaptation Manager is responsible for "executing" the respective configuration, generating monitors that collect data, and forwarding it to the Runtime Model. In case the SuM changes as a result of its own adaptation (e.g., a sensor is removed) – or when the user decides to manually adjust the configuration, the Monitor Adaptation Manager recognizes these changes and updates the generated monitors accordingly, e.g., by deactivating certain monitors, adding new ones, or changing the period of which data is forwarded to the monitoring infrastructure. Furthermore, using the state transition diagrams, the Monitor Adaptation Manager can react to changes in the state of the SuM and adjust monitoring parameters when needed.

• **Runtime Model:** The Runtime Model is instantiated using the previously defined SuM Model. This part is responsible for collecting and distributing runtime information. The Runtime Model is needed to effectively connect and map additional services using the runtime data, for example, a constraint engine or a database. Our preliminary work on generic model-based monitoring [17] provides additional details about the runtime model instantiation.

• **Constraint Engine:** Depending on the type of constraint that needs to be checked, different Constraint Engines (e.g., Drools[1] for process-oriented rules, or Complex Event Processing (CEP) for temporal constraints) might be employed. The user can then define domain-specific constraints based on the individual use case and environment in which the SuM operates. This can include, for example, performance constraints, safety-related checks (e.g., ensuring a minimum battery level), or checking that events are executed in the prescribed order. The Constraint Engine then checks if any predefined constraints are violated, for example, by extracting data about the current velocity provided via the Runtime Model and checking if the values exceed a certain speed limit (C5).

• **Runtime Dashboard:** The Runtime Dashboard is used to visualize collected data and provide the user with additional information about the status of the system (C6). This includes information about the currently active parts of the SuM (i.e., where data is currently collected from, and how often), as well as information about constraints and respective violations that have occurred. Based on this information, the user can then decide to intervene, or further configure the monitor – for example, to collect additional data after a violation has been reported.

• **Data Persistence:** Finally, besides evaluating constraints at runtime, our approach supports the collection and storage of data for post-mortem analysis purposes. This enables offline, potentially complex, analysis on event traces and also supports simulations, and replaying of events that resulted in constraint violations [15].

3.2 Prototype Implementation

For assessing the feasibility of the proposed approach, we have implemented key aspects, with a main focus on the Monitor Adaptation Manager, and applied it to a scenario where ROS-based TurtleBot robots are monitored while performing different tasks. Both the System Modeler and Monitor Adaptation Manager are currently implemented in Python. The "rosbridge suite" package is used for connecting to the ROS system and collecting information about the running system (e.g., the battery status, velocity, or diagnostics data of the TurtleBots). This information is used to select and configure the respective properties and map them to the ROS topics that should be monitored. For each selected property, the Monitor Adaptation Manager then generates a respective topic subscription using the rosbridge package. The subscriptions are dynamic and can be activated and deactivated at runtime whereby the frequency determines how often data is forwarded from the Adaptation Manager to the Runtime Model. We generate the Runtime Model using an Eclipse EMF model that represents the SuM. Data transport from the Monitor Adaptation Manager to the infrastructure and from the infrastructure to the different services is realized via MQTT, which is designed for machine-to-machine communication in IoT environments. As both, MQTT and ROS, use a publish/subscribe pattern for communication, ROS topics can be easily mapped to each of the monitored properties. Our prototype has demonstrated that dynamically collecting runtime data, changing collection frequencies on-demand, and providing user feedback can largely be automated and implemented in a generic way independent from the actual SuM. Additionally, for runtime constraint evaluation, we have used Complex Event Processing for defining complex constraints, such as distance checks between bots, and the Viatra Model Query engine for defining state-based checks, such as minimum battery levels. Finally, a web-based dashboard provides visualizations and information to the user showing runtime data and constraint violations.

[1] https://www.drools.org

4 RELATED APPROACHES AND ROADMAP

Runtime monitoring has been an active research area with many different monitoring solutions that support different monitoring needs and types of systems [12]. Several approaches so far have tackled the issue of adaptive monitoring [4] and dynamic checks using runtime models [14]. Some approaches already exist that specifically target ROS-based systems for monitoring certain types of properties [11, 19]. However, unlike existing approaches, our aim is to significantly reduce the human effort of initially establishing runtime monitoring support and then maintaining the infrastructure when the SuM changes. Providing modeling and monitor adaptation support, our initial prototype implementation indicates that the approach can be easily applied to different ROS-based systems and that we can successfully model, monitor, and adapt the system without the need to continuously update and re-implement probes or monitoring components.

Once fully implemented, we are committed to provide our work as an open-source project which will be publicly made available on GitHub. Based on our progress so far, we plan to focus on the following research aspects in our ongoing work:

•*Efficient Data Processing.* Besides activating and deactivating monitors, and changing the frequency of data collection, additional configuration options require more sophisticated data processing (wrt. C4). For example, instead of forwarding data more frequently, data might be aggregated, or certain calculations can directly be performed on the edge device, before being sent to the cloud and the Runtime Model. Depending on the state of the system, and available computation capacities, aggregation strategies might also dynamically adapt at runtime.

• *Resolving Conflicting Monitoring Configurations.* When a human is involved in an adaptive system or intervenes in an automated decision-making process, this inevitably results in conflicts. For example, a user decides to manually decrease the monitoring period for a certain property, e.g., the current position of a robot, but this conflicts with minimum safety requirements for checking constraints. We are therefore planning on providing meaningful and valid configuration options to the user, and providing explanations and justification, whenever specific options are not available or would jeopardize the correct behavior of the SuM.

• *Diverse Constraint Checks.* While our prototype implementation has demonstrated that different constraint engines can be easily integrated (cf. C5), writing and maintaining different types of constraints can quickly become a complex and cumbersome task. A simplified domain-specific language for runtime constraints that is then used to generate CEP rules, OCL constraints, or other more formal rules could help to reduce complexity and learning effort for the user writing and maintaining these constraints.

5 CONCLUSION

In this paper, we have reported our initial efforts towards designing and deploying a flexible and easily configurable monitoring framework for ROS-based systems. As part of our ongoing and current work, we are implementing our framework as a fully self-contained Python application, complemented by a web-based Monitoring Dashboard, that can interact with any ROS-based system and provide configuration templates for constraints and database storage. We are further expanding our application scenarios, working on creating a monitoring solution for the DroneResponse [1] system, a fully autonomous, ROS-based, multi-UAV system.

ACKNOWLEDGMENT

The work in this paper has been funded by the Linz Institute of Technology (LIT-2019-7-INC-316).

REFERENCES

[1] Ankit Agrawal, Sophia Abraham, Benjamin Burger, Chichi Christine, and et al. Fraser. 2020. The Next Generation of Human-Drone Partnerships: Co-Designing an Emergency Response System. *Proc. of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), 1–13.

[2] Giovanni Beltrame, Ettore Merlo, Jacopo Panerati, and Carlo Pinciroli. 2018. Engineering safety in swarm robotics. In *Proc. of the 1st Int'l WS on Robotics Software Engineering*. ACM, 36–39.

[3] Gordon Blair, Nelly Bencomo, and Robert B France. 2009. Models@run. time. *Computer* 42, 10 (2009), 22–27.

[4] Thomas Brand and Holger Giese. 2019. Modeling Approach and Evaluation Criteria for Adaptable Architectural Runtime Model Instances. In *Proc. of the 22nd Int'l Conf. on Model Driven Engineering Languages and Systems*. 227–232.

[5] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. 2014. Diagnosing Unobserved Components in Self-Adaptive Systems. In *Proc. of the 9th Int'l Symp. on Software Engineering for Adaptive and Self-Managing Systems*.

[6] J. Ruby Dinakar and S. Vagdevi. 2017. A study on storage mechanism for heterogeneous sensor data on big data paradigm. In *Proc. of the 2017 Int'l Conf. on Electrical, Electronics, Communication, Computer, and Opt. Techniques*. 342–345.

[7] Didem Gürdür, Aneta Vulgarakis Feljan, Jad El-khoury, Swarup Kumar Mohalik, Ramamurthy Badrinath, Anusha Pradeep Mujumdar, and Elena Fersman. 2018. Knowledge Representation of Cyber-physical Systems for Monitoring Purpose. *Procedia CIRP* 72 (2018), 468–473.

[8] David B. Lindenmayer and Gene E. Likens. 2009. Adaptive monitoring: a new paradigm for long-term research and monitoring. *Trends in Ecology & Evolution* 24, 9 (2009), 482–486.

[9] Lola Masson, Jérémie Guiochet, Hélène Waeselynck, Augustin Desfosses, and Marc Laval. 2017. Synthesis of Safety Rules for Active Monitoring: Application to an Airport Light Measurement Robot. In *2017 First IEEE International Conference on Robotic Computing (IRC)*. 263–270. https://doi.org/10.1109/IRC.2017.11

[10] Christoph Mayr-Dorn, Mario Winterer, Christian Salomon, Doris Hohensinger, and Rudolf Ramler. 2021. Considerations for using Block-Based Languages for Industrial Robot Programming-a Case Study. In *Proc. of the 2021 IEEE/ACM 3rd Int'l Workshop on Robotics Software Engineering*. IEEE, 5–12.

[11] Samuel Parra, Sven Schneider, and Nico Hochgeschwender. 2021. Specifying QoS Requirements and Capabilities for Component-Based Robot Software. In *Proc. of the 3rd Int'l WS on Robotics Software Engineering*. IEEE, 29–36.

[12] Rick Rabiser, Klaus Schmid, Holger Eichelberger, Michael Vierhauser, Sam Guinea, and Paul Grünbacher. 2019. A domain analysis of resource and requirements monitoring: Towards a comprehensive model of the software monitoring domain. *Information and Software Technology* 111 (2019), 86–109.

[13] Rick Rabiser, Jürgen Thanhofer-Pilisch, Michael Vierhauser, Paul Grünbacher, and Alexander Egyed. 2018. Developing and evolving a DSL-based approach for runtime monitoring of systems of systems. *Automated Software Engineering* 25, 4 (2018), 875–915.

[14] Lucas Sakizloglou, Sona Ghahremani, Thomas Brand, Matthias Barkowsky, and Holger Giese. 2020. Towards Highly Scalable Runtime Models with History. In *Proc. of the IEEE/ACM 15th Int'l Symp. on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 188–194.

[15] Jürgen Thanhofer-Pilisch, Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. 2016. Event capture and compare for runtime monitoring of systems of systems. In *Proc. of the 1st Int'l WS on Var. and Complexity in Software Design*. IEEE, 1–4.

[16] Michael Vierhauser, Sean Bayley, Jane Wyngaard, Wandi Xiong, Jinghui Cheng, Joshua Huseman, Robyn Lutz, and Jane Cleland-Huang. 2019. Interlocking Safety Cases for Unmanned Autonomous Systems in Shared Airspaces. *IEEE Transactions on Software Engineering* 47, 5 (2019), 899–918.

[17] Michael Vierhauser, Hussein Marah, Antonio Garmendia, Jane Cleland-Huang, and Manuel Wimmer. 2021. Towards a Model-Integrated Runtime Monitoring Infrastructure for Cyber-Physical Systems. In *Proc. of the 2021 IEEE/ACM 43rd Int'l Conf. on Software Engineering: New Ideas and Emerging Results*. IEEE, 96–100.

[18] Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. 2016. Requirements monitoring frameworks: A systematic review. *Information and Software Technology* 80 (2016), 89–109.

[19] Thomas Witte and Matthias Tichy. 2021. Inferred Interactive Controls Through Provenance Tracking of ROS Message Data. In *Proc. of the 3rd Int'l WS on Robotics Software Engineering*. IEEE, 67–74.