

GUIDELINES

In the appendix we describe two sets of guidelines: guidelines for enabling and facilitating runtime verification and field-based testing for ROS-based applications through development are presented in Sect. 1 and guidelines for providing quality assurance through runtime verification and field-based testing for ROS-based applications are detailed in Sect. 2.

1 GUIDELINES FOR DEVELOPMENT TO SUPPORT RUNTIME VERIFICATION AND FIELD-BASED TESTING

In this section, we detail eight guidelines for facilitating the verification of ROS-based applications through development activities. Our approach to facilitate verification lies on providing guidance to robotics software developers. The group of Constraint Identification (CI) guidelines contains three guidelines, CI1, CI2, and CI3, devoted to identifying constraints of different nature, i.e., timing constraints, security and privacy constraints, and safety constraints. The group of Code Design and Implementation (CD) guidelines contains two guidelines, CD1 recommending to design ROS nodes with single responsibility and CD2 guiding towards ensuring global time monotonicity of events and states. The last group contains four Instrumentation (I) guidelines. I1, I2, and I3 focus on providing APIs for querying and updating an internal lifecycle (I1), for logging and filtering (I2), and for injecting faults in execution scenarios (I3). The last one (I4) focuses on isolating components for testing.

1.1 CI1. Identify timing constraints

1.1.1 Context (WHEN)

Timing constraints, also known as “deadline” or “timing requirements”, define temporal requirements that real-time systems must adhere to [1]. These constraints can be categorized as soft, firm, or hard, and they play a significant role in determining whether the system respects timing concerns. Since ROS-based systems often involve control nodes (such as attitude control and state estimation [2]), which typically have real-time requirements, identifying timing constraints becomes even more critical.

1.1.2 Reason (WHY)

Identifying timing constraints is crucial for ROS-based systems as these constraints define the system’s temporal requirements. However, a recent study highlights the lack of support for real-time constraints in the software engineering research of ROS-based robotic systems, emphasizing the need for more attention in this domain [3]. Initially conceived as a means to handle difficult or costly-to-reproduce failures, field testing has seen limited studies addressing real-time issues, particularly in the realm of autonomous systems [4].

1.1.3 Suggestion (WHAT)

The development team should identify timing constraints to ensure that no real-time requirements will be neglected during the system testing. For instance, Autoware_Perf (git:azu-lab/ROS2-E2E-Evaluation) allows for the calculation of response time and latency in ROS 2 applications; such measurements may be used as hard constraints to testing



the system. Also, one may identify real-time constraints with respect to synchronization between robots, sensor data processing time, or fault detection and recovery.

1.1.4 Process (HOW)

Typical timing constraints that cross-cut in robotics domains stand for response time [5], latency, synchronization time [6], fault detection and recovery time, and power management time. The means to specify such constraints vary from informal to formal languages (we invite the interested reader to check guidelines of the group property specification - SDB1 and SDB2. The developers team may use Autoware_Perf [7] to calculate response time and latency, for example.

1.1.5 Exemplars

Typical Timing Constraints. Real-time constraints may be subject to the domain and the objectives of the runtime assessment. We provide a comprehensive list of constraints that may be interesting to address when considering testing ROS-based systems. We use Autoware_Perf [7] to illustrate how to collect response time and latency with practical examples.

- **Response time** is the period a ROS-based system takes to react to one or more stimuli from the environment. Specific to the ROS domain, Chaaban, K. defines response time in ROS 2 as the “*duration from the release instant to the completion of the job execution*” [5]. To collect callback response time, Autoware_Perf defines a function to get durations of callback instances for a given callback objects (Autoware_Perf/./callback_duration.ipynb .
- **Latency** is the time a ROS-based system takes to react to a stimulus from the test orchestrating apparatus. Often, field tests include a centralized computer orchestrating the test cases execution and information collection, latency in such cases tends to be significant. Autoware_Perf illustrates how latency can be computed with an example of a ROS 2 service, where they send a goal to the robot and calculate the time for a response (Autoware_Perf/./e2elatency.ipynb .
- **Synchronization time** is the time taken for a ROS-based system involving more than one robot to share information.
- **Sensor Data Processing time** is the time taken to transform raw sensorial input into information ready to be used by reasoners or planners. Lewis et al. conclude that it is difficult to predict how long robots might have to wait between measurements since quantifying data collection for real-time systems needs to account for all the [metereological] sensors and their respective response times[6]
- **Fault Detection and Recovery time** is the time taken to prevent safety hazards. Robots, mainly operating in safety-critical scenarios, often have safety mechanisms to prevent hazards. Testing in the field, in general, asks for safety assessment, thus incurring timing overhead.
- **Power Management time** is the time taken to prevent stop or graceful degradation due to a power outage.

Robots may rely on batteries or other embedded power sources (such as solar panels). Power management may affect the testing conditions and be considered when testing in the field. For example, to start a recharging process in the middle of the mission or to slow the velocity to recharge the batteries using solar energy.

1.1.6 Strengths

Identifying timing constraints is a good practice to avoid neglecting important real-time requirements during development. It also helps isolating the tested behavior without unintended outcomes due to real-time faults.

1.1.7 Weaknesses

Specifying timing constraints may be complex due to the ROS nature presenting interconnected components and dependencies. There is a lack of standardization or best practices for specifying such properties. This may result in an overhead when the application is not real-time critical. Property specification guidelines (SDB1, SDB2) will further investigate this issue.

1.2 CI2. Identify security and privacy constraints

1.2.1 Context (WHEN)

Runtime assessment introduces challenges to security and privacy preservation. Modifying the source code to facilitate testing may, however, open space for security and privacy exploitation[4]. From the point-of-view of runtime verification, security and privacy preservation revolves around specifying flavors of integrity, confidentiality, and availability[8]. For instance, Autonomous Driving Vehicles (ADVs) regulators define policies given the territory they are being used in. However, ADS introduction also requires public trust, w.r.t security and safety [9]. In a scenario of runtime assessment, be it field-based testing or runtime verification, security and privacy exploits may be catastrophic. Yet, security is still under-explored when it comes to architectural designs for ROS-based applications [10].

1.2.2 Reason (WHY)

Security and Privacy are often under-explored in the design of ROS-based applications. Threats to privacy and security may be catastrophic and may be an unacceptable side-effect of runtime assessment of such robotic applications.

1.2.3 Suggestion (WHAT)

The development team should identify security or privacy vulnerabilities that may pose a risk to participants' integrity, confidentiality, and availability, that may be caused by the testing and runtime verification activities. The ros-security workgroup, in collaboration with Alias Robotics, maintains SROS (git: ros2/sros2) for the analysis of such vulnerabilities and fixes. In addition, Alias Robotics maintains a database of vulnerabilities detected in ROS applications (git: aliasrobotics/RVD) as a means of awareness.

1.2.4 Process (HOW)

Jeong S. Y. et al. [11] list a set of vulnerabilities typically encountered in ROS applications, including broken authentication, the rosbag replay attack, and service hijacking. Neither preparing the code for runtime assessment nor the runtime assessment procedure itself should enable or facilitate the exploitation of such vulnerabilities. The developers must be aware of such vulnerabilities and specify constraints specific to the domain of the application, enforcing or informing the testing of the danger. From one side, the ros-security working group works on mitigating security vulnerabilities provided by design from ROS [12]. From another stance, other approaches focus on confidentiality, integrity, and availability while providing means to specify control access policies and generate firewall rules based on the fundamental elements of ROS [13], [14].

1.2.5 Exemplars

FAST-DDS is the current technology used in ROS 2¹. FAST-DDS enables features such as authentication of domain participants using a Certificate Authority (CA), access control to constrain specific operations, authenticated encryption following the Authentication Encryption Standard (AES), logging security events, and data tagging by enabling security labels to data. In addition, the ros-security working group, in tandem with Alias Robotics, works on SROS, a tool for securing ROS within the DevOps model (ros2/sros) [12].

Hu et al. specify security properties of ROS-based systems [13]. The specified security properties follow the definition of a composite of attributes on confidentiality, integrity, and availability, according to Avizienis [15]. Hu et al. define the properties following two scenarios: (i) attackers get access to a ROS node and freely allocate memory until reaching an out-of-memory state, (ii) topics or services are hijacked, and the information flowing between ROS nodes is changed to provoke a failure. In such scenarios, the developer's team should specify constraints to the runtime assessment procedure placing a barrier to facilitating the control of ROS nodes in such a way that enables free memory allocation and should specify conditions to allow ROS topics and services remapping without any certification, for instance. The testing team may want to assess whether their runtime assessment will open up exploit opportunities given a set of constraints.

ROSRV, a runtime verification framework for safety and security properties of ROS-based applications, provides a specification language for access control policies and enforces them at runtime [14]. Such access control policies may prevent the system from exploiting security breaches, such as enabling the control of safety-critical nodes to the application. More specifically, ROSRV provides access control specifications based on a user-provided specification of access policies using text file inputs. The policies are defined under five categories: groups, nodes, publishers, subscribers, and commands. Nodes represent node names and machine identities, publishers/subscribers define topic names and node identity associated, and commands define the access policy, e.g., full access, localhost (ROSRV).

1. Details on FAST-DDS <https://fast-dds.docs.eprosima.com/en/latest/fastdds/security/security.html>

1.2.6 Strengths

Identifying security and privacy constraints can help protect participants and resources during runtime assessment. This guideline encourages developers to consider security and privacy early on in the design process, rather than as an afterthought. It encourages the development of more secure and trustworthy ROS-based applications and runtime assessment of ROS-based applications, which can improve public trust and acceptance.

1.2.7 Weaknesses

The guideline, intentionally, does not provide specific details on how to implement security and privacy constraints, leaving it up to the developers to decide which constraints are necessary and how to enforce them, since this is domain-dependent. It may be challenging to understand the various security vulnerabilities associated with ROS-based applications, making it difficult to specify appropriate security and privacy constraints. The guideline may require additional time and resources to implement security and privacy constraints, which may delay the development process. Systems with lower technology readiness level, such as academic prototypes, may not need a thorough security assessment, making this guideline irrelevant in those cases.

1.3 CI3. Identify safety constraints

1.3.1 Context (WHEN)

The assessment of safety constraints is essential for both runtime verification and field-based testing. When stimulating robotic systems with extreme scenarios, for instance in robustness testing [16], it's crucial to consider potential threats to personnel and the properties being tested. These threats limit the ability to conduct online assessments for safety-critical applications, particularly in robotics. To ensure the benefits of field-based testing [4], it's important to carefully address and compensate for the introduced safety risks. Non-negotiable safety constraints should be maintained throughout the confidence-gathering process, as they determine the readiness and release of technology, both in industry and everyday use [17].

1.3.2 Reason (WHY)

Safety hazards may be unable to be verified by runtime verification or field-based testing since corner cases may pose threats to testing personnel or damage property during runtime assessment.

1.3.3 Suggestion (WHAT)

The developers should identify the boundaries of a safe behavior (i.e., safety constraints) to enable the use of mechanisms for preventing the robot from hurting operators or property during testing and verification activities. For example, the developers can identify constraints like speed limit, distance to obstacle, or conditions for emergency stop, which can be used by monitoring tools like ROSMonitoring (git: [autonomy-and-verification-uol/ROSMonitoring](https://github.com/roboticstudies/autonomy-and-verification-uol)), or used by an independent safety controller in a separate ROS node to prevent the robot from falling from a cliff, e.g., the Kobuki robot (git: [yujinrobot/kobuki](https://github.com/yujinrobot/kobuki)).

1.3.4 Process (HOW)

Safety constraints may be specified in using many specification languages. The known toy example in mobile robotics, Kobuki, specifies safety properties in a separate ROS node using the switch-case in Python; Adam et al. [18] proposes rule-based specifications in the format of boolean equations that predicate over messages over topics; Stadler et al. [19], uses an event processing language offering query-based operations; Finally, Huang et al. [14] enables the specification of temporal properties leveraging monitoring-oriented programming (MOP).

1.3.5 Exemplars

Typical safety constraints:

- emergency stop [18]
- speed monitoring (e.g., $v < 0.2m/s$) [18], and movement speed limit (e.g., speed limit $0.35m/s$) [19]
- timing monitoring (e.g., $t > 10$) [18]
- obstacle avoidance (e.g., maintain at least $35cm$ distance from object) [19]
- joint effort (e.g., `max_joint_effort`) [19]

Technologies:

The Kobuki robot implements a safety controller (🔗). The Kobuki's safety controller specifies, using Python at code-level, a state machine ensuring that the robot stops whenever it is close to a cliff, the bumper sensor was pressed, or at least one of the wheels dropped (or was raised in the air). The Kobuki's safety controller is a separate ROS node collecting bumper, cliff and wheel events. A sequence of events may trigger an alert or issues velocity commands to deviate Kobuki from the threat.

Adam S. et al. [18] defines a rule-based language for enforcing safety constraints on existing ROS-based software. Their approach uses a simplified metamodel to describe the ROS software enabling static analysis as well as invariant monitoring. The authors explain that separating the specification of safety issues from functionality facilitates attesting that the robotic behavior conforms to safety requirements. Their rule-based approach enables the specification of actions that preclude rules encoded in boolean equations (e.g. `cmd_vel_left > 0.02m/s`) over messages in topics.

Stadler et al., [19] uses the Esper Constraint Engine to check for violations of the safety constraints. Such constraints are specified using the EPL notation. Event Processing Language (EPL) is a SQL-standard language with extensions, offering SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses. For instance, they specify that an event such as JointEffortEvent (AEvent) will not exceed a maximum of 5.0 in an ESL constraint specification (i.e., `max_joint_effort`).

Listing 1: Safety constraint in EPL notation.

```
select * from AEvent (MaxValue > 5.0)
```

Huang et al.'s [14] monitoring infrastructure enables the use of any logic plugins of Monitoring-Oriented Programming, thus enabling specification of temporal properties over events, such as regular expressions, linear temporal

logics, and context-free grammars. For instance, one of their monitors predicates on the position of the turret installed in the autonomous vehicle, i.e., “the gun should not exceed a maximum joint angular position otherwise the turret targets itself.”.

1.3.6 Strengths

By identifying safety constraints, developers can demonstrate compliance with these requirements, providing assurance to stakeholders that appropriate safety measures are in place. Early detection and response to safety violations, such as exceeding speed limits or approaching obstacles too closely. This early detection and response can help prevent accidents and minimize potential damage.

1.3.7 Weaknesses

Identification of safety hazards involves understanding the system’s behavior, identifying potential risks, and formulating appropriate constraints. This process can be time-consuming and complex, especially for large and intricate systems, potentially increasing development costs.

1.4 CD1. Developers should strive for ROS nodes with single responsibility

1.4.1 Context (WHEN)

ROS fosters a rich toolkit for designing modular robotics software [20]. Developers face design decisions of how to implement a given set of requirements using ROS fundamental concepts, i.e., nodes, topics, services, packages [10]. In order to facilitate runtime assessment, the system’s computational units should be designed such that they can be safely exposed to trials with warranted no side effects on the running system [4]. In addition, it should reduce the cost of observing and controlling modules, or, in robotics, atomic actions (or skills) [21].

1.4.2 Reason (WHY)

Mapping functional units to ROS nodes enables fine-grained observation and control of the robotic behavior.

1.4.3 Suggestion (WHAT)

To facilitate assessment during the robot operation and enable fine-grained observation and control, the developers should implement ROS nodes following the single responsibility principle (i.e., each node should implement a single feature and different nodes can be combined to perform a complex task). For example, developers should implement independent nodes for path planning and reactive manoeuvring, or independent nodes for defining primitive skills like grasping an object or simultaneous localization and mapping (SLAM).

1.4.4 Process (HOW)

We divide the means to designing ROS nodes with single responsibility in two: hierarchical design, and skill-based design. In hierarchical design, ROS nodes are organized based on the goals they achieve, with each node responsible for a specific task or functionality [22], [23]. This approach allows for clear separation of responsibilities and easier

maintenance. In skill-based design, ROS nodes are organized based on the skills they possess, with each node responsible for a specific capability or behavior [24], [25], [26]. This approach enables more flexibility and reusability of nodes, as they can be combined to perform various tasks.

1.4.5 Exemplars

Hierarchical design. Hartswell et al. define components as building blocks that may be hierarchically composed to fulfill the system’s functional requirements [22]. Their running example implements ROS nodes with complementary but distinct, and independent, functionalities. One node resolves path planning in a deliberative control setting, and the other is responsible for the low-level PID controller reactively manoeuvring the robot. In the healthcare domain, the Body Sensor Network (lesunb/bsn²) defines ROS nodes after concrete tasks that, composed, should fulfil the system’s goal. Tasks such as collecting SPO2 data, collecting heartbeat data, fuse data, or identifying emergency [23]. For instance, the collect SPO2 data task implements an independent node for collecting, filtering, and transferring data ³. Given that each sub-task (i.e., collecting, filtering, and transferring) is strictly dependent on each other, yet, should not depend in the environment and together form an atomic behavior. Separating the sub-tasks in different nodes may render incomplete test cases that are hard to read and often meaningless.

Skill-based design typically defines skills as fundamental software building blocks operating a modification on the world stage. In such approaches, skills should be modular and reusable. Although ROS nodes provide a natural means for modular design, it is not common to map skills to nodes in current skill-based designs. For instance, SkiROS [24] implements 3 nodes: World Model Manager, Task Manager, Skill Manager (skiros2_skill⁴). In addition, HRMS [25] does not map skills to nodes. It builds skills that under the Behavior Trees notation may reuse existing nodes (hmrs/skills⁵). Lack of standard mapping from skills to actual computation impairs the verification of skill-based approaches at skill level. Towards verifiable ROS-based applications, a promising architecture, namely RobMoSys², suggests separation of concerns between different levels such as Mission, Task Plot, Skill, Service, Function, whereas Skill concerns to configuration, e.g., grasp object with constraint, and functions concern to computation, e.g. inverse kinematics solver. In this direction, Albore A. et al. [26] promote ROS2 code generation whilst mapping skillsets to ROS nodes with a single responsibility.

1.4.6 Strengths

Modularity enables fine-grained observation of system requirements. In addition, it caters to clear interfaces for runtime inspection and allows for mocking and substituting behavior for assurance scenarios.

1.4.7 Weaknesses

Feedback loops, however, may impair testing, i.e., modularity turns out to be costly for systems with feedback loops, whereas the system’s behavior is context-dependent [27].

2. <https://robmosys.eu/wiki>

1.5 CD2. Ensure global time monotonicity of events and states

1.5.1 Context (WHEN)

As distributed systems, such as ROS-based applications, rely on time scheduling for their operation, ensuring time monotonicity is crucial to guarantee the replicability and reproducibility of test cases. Non-determinism in the scheduling of events can lead to unexpected behavior, compromising the reliability of tests and hindering their reproduction. To address this issue, developers must ensure that the testing team is aware of the expected behavior given the occurrence of events, which requires establishing a clear execution order and priority. For instance, by synchronizing different types of data at package or subscription-level [10].

1.5.2 Reason (WHY)

Ensuring global time monotonicity of events and states permits to address the potential non-determinism in the scheduling of events in ROS-based applications.

1.5.3 Suggestion (WHAT)


The development team should ensure global time monotonicity of events and states to avoid potential non-determinism in scheduling. Such non-determinism is a threat to getting confidence in the system since repeated tests under the same conditions may turn into different results. A technique that can be used to ensure determinism is annotating messages and requests with timestamps and implementing a logical time synchronizer, similar to what is done by MAVROS (git: mavlink/mavros). Also, the Time Synchronizer message filter (wiki: message_filters) may be used for this purpose.

1.5.4 Process (HOW)



Events are said to be monotonic in time whenever for a set of events scheduled over a global clock, two consecutive events are successive. First the developers should understand how events are scheduled and the limitations of the application towards predictable ordering of events [5], [28]. Then, the developers should analyze and guarantee that all processes are executed in the expected order and will lead to predictable behavior. Developers may accomplish such analysis and guarantee provision with external tools, for instance, formal methods [29], [30] or by embedding timestamps (e.g., SteadyTime) and filtering messages to guarantee ordered events (e.g., MAVROS).

1.5.5 Exemplars

Understand the scheduling of ROS events and states.

Chaaban K. [5] explains that ROS 2 schedules callbacks using a non-preemptive algorithm that consumes messages depending on their type. Unlike typical real-time priority-based scheduling algorithms, ROS 2 does not execute callbacks in their activation instances. Thus non-time-based messages are scheduled in a round-robin fashion. To this extent, ROS 2 (which claims to address real-time) presents limitations when asked for well-defined execution orders and priority inversion. Choi et al. [28] promotes rtenlab/ros2-picas , a new scheduling algorithm that adds priority awareness for scheduling in ROS2.

Analyse starvation-freedom to support monotonicity. Bläß T. et al. [29] proposes a response-time analysis exploiting the round-robin properties. The authors present an analysis of starvation-freedom using three techniques: modeling processor demand as execution-time curves instead of scalar worst-case execution times (WCETs), accounting for the effects of quiet times and busy windows in the activation curve derivation, and exploiting the round-robin behavior of the ROS callback scheduler. Starvation freedom indicates monotonicity and ordering of events, it helps the developers to understand real-time behavior and set expectations. Halder R. et. al [30] use model checking (e.g., UPPAAL [31]) to analyze real-time behavior in ROS-based applications. Under the claim that ROS non-deterministically empties the communication queues, the authors focus on modeling the message-passing behavior of ROS and verifying whether the queue overflows. Since overflow here means that some process is starving, thus the application is not monotonic in time. The authors offer the models in timed automata and TCTL (Temporal Computational Tree Logic) properties designed to ensure real-time for the safety controller of the Kobuki³ exemplar.

Annotating messages with timestamps and synchronization to guarantee ordering. The SteadyTime⁴ class in ROS represents a monotonic clock, functioning independently of physical time and with steady tick times. MAVROS (mavlink/mavros ) , a library broadly used in ArduPilot for flight control, approaches time monotonicity guarantees by implementing a logical time synchronizer. The synchronizer ensures that the time passing in MAVROS is aligned with the time on ArduPilot's side, avoiding drifts that could lead the application to fail. More importantly, the time synchronizer uses a particular type of message that embeds timestamps for clock synchronization for updating ArduPilot about its time status (TimesyncStatus.msg ) . Messages such as this can be used for synchronizing the occurrence of events in the system. The Time Synchronizer message filter⁵, for example, reads timestamps from the header of messages and synchronize message by outputting channels to a single callback.

1.5.6 Strengths

Tests reliability increases by enabling the prediction of the expected behavior of a system under scrutiny. In addition, time monotonicity guarantees can facilitate reproducing test results by establishing a clear execution order and priority. Finally, developers can reduce the time taken and resources used to verify the system behavior by implementing time synchronization and annotation messages with timestamps.

1.5.7 Weaknesses

Implementing time synchronization to ensure time monotonicity may require additional development effort, adding complexity to the system. Also, time synchronization may

3. <http://wiki.ros.org/kobuki>

4. https://design.ros2.org/articles/clock_and_time.html

5. https://wiki.ros.org/message_filters

add performance overhead due to additional operations execution. Achieving real-time behavior depends on the (interaction with the) operating system.

1.6 11. Provide an API for querying and updating internal lifecycle

1.6.1 Context (WHEN)

Autonomous systems, such as robotic applications, often require stateful compute nodes [27]. However, the internal states within ROS nodes are typically hidden and not easily accessible, limiting the ability to diagnose and understand unexpected behavior. Exposing these hidden states is crucial for providing granular information on the system's behavior, rendering increased observability. Furthermore, managing these hidden states allows for actions such as starting, stopping, and rolling back to a specific state, in other words, increased controllability. The running system should be both observable and controllable [8], [4], as these states can be application-specific and not standard [10].

1.6.2 Reason (WHY)

The use of ROS nodes with lifecycle management can facilitate testing by providing a structured way to manage the state of the nodes and the interactions between them. This structure helps to ensure that nodes are in the right state for testing and that the interactions between nodes are predictable. Furthermore, it helps mitigate dangling references to nodes that are no longer in use.

1.6.3 Suggestion (WHAT)

To facilitate field-based testing, the development team should properly manage the ROS nodes' lifecycle and prepare APIs for querying and updating the internal nodes' life-cycle, e.g., to ensure that nodes are in the right state for testing. For example, developers can use modes (git: ros2/demo) for lifecycle management. In the context of Micro-ROS (git: micro-ROS), developers can define custom lifecycle modes (git: micro-ROS/system_modes) like sleep, power saving, starting, processing, and ending modes, and use separate extra nodes for mode monitoring and mode update.

1.6.4 Process (HOW)

The team should define a custom life-cycle for the ROS-based application, for instance, the life-cycle phases could be: starting, processing, and ending [32], using system_modes from Micro-ROS[33]. Moreover, the processing phase can further be divided into waiting and executing steps and application-specific states, and the transitions between phases should be well-defined by events. The development team should then prepare an API for managing the life-cycle of nodes [34], which can be queried or updated through ROS interfaces, such as the client-server interface. The API should allow for the representation of all nodes in the application, which can be queried or updated using the interface.

1.6.5 Exemplars

Customized life-cycle Conte G. et al. [32] define a custom life-cycle in ROS for their autonomous surface vehicle. The

life cycle has three phases: starting, processing, and ending. The processing phase can be subdivided into two other: waiting and executing. Events define the transition between phases. The authors claim that such organization helped in the field testing when they had to explicitly shut down a (ROS) agent, shut down the agency infrastructure, or detect a failure in the ROS infrastructure. ROS2 provides the concept of life-cycle by design. Developers may inherit from the LifecycleNode from rclcpp to enable the standardized life-cycle management from ROS2. Belsare K. et al. [33], namely Micro-ROS, extends the ROS2 life-cycle to the domain of embedded systems. Their extension enables the specification of (sub-)systems or systems-of-systems in a hierarchical finite-state machine style. It creates the concept of modes within the active state. System modes (micro-ROS/system_modes) are customizable and can be used to track sleep modes and power saving [34]. The authors present an example (micro-ROS/system_modes/system_modes_examples) of how can add *weak* and *strong* as customized modes for an active manipulator.

Listing 2: YAML definition of custom system modes weak and strong for torque control.

```
manipulator:
  ros_parameters:
    type: node
    modes:
      __DEFAULT__:
        ros_parameters:
          max_torque: 0.1
      WEAK:
        ros_parameters:
          max_torque: 0.1
      STRONG:
        ros_parameters:
          max_torque: 0.2
```

Life-cycle management is the ability to query and update the nodes' life-cycle. rcl/rcl_lifecycle the library used by ROS2 provides the functionalities to managing nodes using the ROS2 stack [35]. rcl_lifecycle implements a ROS node that represents all other ROS nodes within the application. The nodes encode finite state machines that can be queried or updated using the ROS service interface. The lifecycle_talker example (ros2/demos/lifecycle) illustrates how the get_state and change_state services query or update the life-cycle. The ROS2 implementation, however, does not support hierarchical states. To mitigate the problem, Micro-ROS defines two extra nodes, the mode monitor node and the mode manager node [34]. The mode monitor infers the state (i.e., mode) of underlying nodes; The mode manager may use the (inferred) state to update modes of sub-systems, modes of nodes, and node parameters.

1.6.6 Strengths

Following this guideline results in reliable and repeatable runtime assessment since the life-cycle manager can ensure that the application is in the desired state. Moreover, it is easier to setup and teardown testing harnesses since it would be possible to understand the current state of the application,

avoiding interruptions in critical phases, and planning for start and stop in the appropriate time.

1.6.7 Weaknesses

As a drawback, there will be overhead associated with meta-data for life-cycle management, and increasing the number of lines of code may impact the system's latency. Also, custom states may ask for larger models representing the possible life-cycle and management, incurring in cost of modeling and maintenance.

1.7 12. Provide an API for logging and filtering

1.7.1 Context (WHEN)

Dynamically gathering information is a fundamental step for gaining confidence in ROS-based systems. Logging (and playback) is, in fact, one of the most used techniques for testing ROS-based systems [16]. Often named *monitoring* [8], or *logging* [4], the process of recording textual or numerical information about events of interest may be a valuable input to the testing team. With such data in hand, the testing team will process the data and transform it into useful information to challenge their hypotheses about how the system should work.

1.7.2 Reason (WHY)

Logging important events depends on instrumenting the code (with 'hooks') that enables the information retrieval activity. It is unrealistic to assume that the testing team will have access to the source code or that the testing team knows what events to log or how to do so.

1.7.3 Suggestion (WHAT)

The development team should provide an API for logging and filtering data to enable access to valuable runtime data which should be used for both runtime verification and field-based testing. The standard approach to logging and filtering is `rosbag` (wiki: `rosbag`). Though, in addition, AWS CloudWatch (git: `aws-robotics/cloudwatchlogs-ros2`) collects data from the `rosout` topic and provides a filter for eliminating noise from the logged events. Another example is the Robotic Black Box (git: `ropod-project/black-box`) which allows for listening to data traffic from distinct sources and logging the messages using MongoDB.

1.7.4 Process (HOW)


We divide, according to Falcone et al. [8], techniques for gathering information in two: inline, and outline. Inline logging and filtering stand for techniques that ask for actually inserting snippets of code in the system under scrutiny as a means to provide an API for logging, e.g., [36]. Outline logging and filtering stands for techniques that enable an external means to gather and filter information that does not require changing the source code, e.g., [37], [38], [39]. Developers may choose one or another given their domain of application.

1.7.5 Exemplars

Inline logging and filtering. ROS, by standard, contains a system-wide string logging mechanism, namely ROS logging (<http://wiki.ros.org/logging>). ROS logging works with a set of macros defined for instrumenting the nodes with information hooks. In the background, the macros send messages with the information to be logged through a standard topic called `rosout`. On the other side of the topic, an extra node, within the `roscore` package, persists the data in a textual format (`ros/./rosout` 📄). Developers can use the macros to log information that might be used for testing in a later stage, given the application-specific requirements. For instance, the Amazon AWS service for robotics provides AWS CloudWatch Logs (`aws-robotics/cloudwatchlogs-ros2` 📄) interfaces directly with `rosout` to monitor applications using the standard ROS logging interface. In addition, the standard library provides logging macros with embedded filtering capabilities, which enables eliminating noise from the logged events and can render a useful tool for testers. ROS Rescue [36] is another example of inline logging. The tool aims at solving the problem of ROSMaster as a single point of failure. The authors approach check-pointing and restoring state by logging changes in the metadata stored in the master node. Such metadata contains URIs from various nodes, port numbers, published or subscribed topics, services, and parameters from the parameter server. Kaveti P. et al. create an API for the ROS master node (`master_api.py` 📄) using the official logging library from Python⁶ to persist metadata in YAML format. Their technique opens space for further inspection of ROS applications without access to the source code.

Outline logging and filtering. ROSMonitoring [37] employs monitors to persist events in textual format. The monitors contain filtering capabilities to eliminate entries that are inconsistent with the specification (requires an oracle). In this context, the launch files to configure ROSMonitoring can be seen as an API for logging and filtering (`ROSMonitoring/./online_config.yaml` 📄). Monitors in ROSMonitoring are nodes, thus, the API is a set of known topics and message formats. The tester, in that case, only needs to specify what topics ROSMonitoring will listen to and the type of message to be recorded. The logging and filtering happen within a separate service. On a similar stance, and inspired by aircraft black box (and software black box [38]), Mitrevski, et al. [39] proposes the concept of Robotic Black Box (`ropod-project/black-box` 📄). The black box operates as an isolated component responsible for listening to data traffic from distinct sources and logging in an easily retrievable medium. Similarly to ROSMonitoring, Mitrevski's black box approach to logging (`ropod-project/black-box/./logger_main.*` 📄) inspects topics and message types that are configured in advance. Different from ROSMonitoring, Robotic Black Box offers logging not only in textual format but also in a MongoDB database, which is essentially useful for data processing, filtering, and retrieval. Robotic Black Box stands out when it comes to its filtering and retrieval capabilities. The approach builds a customized query interface over the MongoDB database using names

6. <https://docs.python.org/3/howto/logging.html>

of collections, timestamps and metadata to filter the results (black_box_tools/db_utils.py )

1.7.6 Strengths

An API for retrieval and filtering facilitates access to valuable information resulting in effortless observability of inner states and events.

1.7.7 Weaknesses

Overuse of logging may result in performance issues. Incorrectly implemented logging and filtering capabilities may lead to noise in the data, impacting the reliability of the tests. Finally, insufficient logging may result in an incomplete assessment of the system behavior and might generate false positives.

1.8 13. Provide an API for injecting faults in execution scenarios

1.8.1 Context (WHEN)

Stimulating ROS-based systems with unexpected scenarios is a cornerstone of gathering confidence in the system's robustness [16], [27]. Such unexpected scenarios either emerge from faults in the robot's control logic or unforeseen inputs unfolding from complex interactions with the environment. Therefore, preparing the ROS-based systems to systematically explore unintended behavior and complex environmental interactions becomes important in runtime verification [8] and field-based testing [4] for robust ROS-based applications. Fault injection and error emulation are the two fundamental approaches to systematically stimulating a computing system with unexpected scenarios for confidence-gathering purposes [40].

1.8.2 Reason (WHY)

Providing an API for stimulating unexpected scenarios for testing the robustness of the target system can help address challenges in finding both fault and error sets representative of real software faults.


1.8.3 Suggestion (WHAT)

To enable runtime verification and field-based testing, developers should provide an API for injecting faults or emulating runtime errors. For instance, to emulate the consequences of software faults, `ros1-fuzzer` (git: `aliasrobotics/ros1_fuzzer`) provides an API for ROS messages fuzzing. As another example, to imitate the mistakes of programmers, IM-FIT (git: `cembglm/imfit`) is a tool for injecting faults in ROS applications.

1.8.4 Process (HOW)

There are two identified means to enabling unexpected scenarios injection in ROS: providing an API fault injection and or an API for error emulation. The two methods differ concerning their assumptions. Fault injection is about changing the source code and requires knowledge about the system's internal structure (e.g., `imfit` [41]). Error emulation is about changing the system during execution and should not require knowledge about the system itself (e.g., `ros1_fuzzer`, and `ros2_fuzzer`). Thus, from the developer's point of view, enabling faulty scenarios through an API requires deciding whether fault injection or error emulation is the most suitable based on the system's requirements.

1.8.5 Exemplars



Fault injection. Fault injection imitates the mistakes of programmers by changing the system under scrutiny's source code [42]. Developers can embed an interface for enabling source code modifications to inject faults in the ROS-based system. For instance, IM-FIT(`cembglm/imfit` ) is a tool tailored to mutation-based software fault injection for ROS applications [41]. Listing 3 describes a subset of the IM-FIT ROS mutations library which offering three operations (delete, change, and external) on defined ROS primitives, such as publishers, subscribers, timers, global variables, namespaces, service clients. Moreover, IM-FIT offers an interface to decide when, where and how the mutators will inflict faults in the source code. Embedding IM-FIT in the project as an API to fault injection facilitates robustness testing for the QA team.

Listing 3: Snippet of the IM-FIT's JSON library containing mutations for fault injection in ROS-based systems.

```

1 {"all_faults": [
2   {"fault": {
3     "Fault_Name": "Subscriber Delete",
4     "Target_to_Change": "rospy.Subscriber",
5     "Changed": " ",
6     "Explanation": "Subscriber deletes"
7   }},
8   ...
9   {"fault": {
10    "Fault_Name": "Periodic Timer Change",
11    "Target_to_Change": "rospy.Timer",
12    "Changed": "rospy.Time",
13    "Explanation": "Periodic timer is wrong"
14  }},
15  ...
16  {"fault": {
17    "Fault_Name": "External ROS Message",
18    "Target_to_Change": "[rospy.Message]{13}",
19    "Changed": "rospy.Message rospy.Message",
20    "Explanation": "External ROS Message"
21  }}
22 ]}

```

API for error emulation. Error injection attempts to emulate the consequences of software faults by manipulating the runtime states/events of the system under scrutiny [43]. Developers can embed an interface within the ROS-based system enabling a set of operations to disturb it during runtime. For instance, `ros1-fuzzer` (`aliasrobotics/ros1_fuzzer` ) provides an API for ROS messages fuzzing. The API requests the topic's name, the message type and datatype strategies to fuzzing the message's content. Although the datatype and constraints are hard coded in the API, defining the topic name and message type are done in the command line. For example, `$ ros_fuzzer -t /rosout -m rosgraph_msgs/Log`, in which the user is fuzzing a `Log` message sent to the `/rosout` topic. The authors evolved `ros1-fuzzer` to suit ROS 2 applications, resulting in `ros2-fuzzer` (`aliasrobotics/ros2_fuzzer` ). `ros2-fuzzer` extends the interface enabling service-level fuzzing, not only message fuzzing as was `ros1-fuzzer`. The addition affected the API asking for an extra argument (`service` or `message`) in the command, e.g., `$ ros2_fuzzer.py service example_interfaces/AddTwoInts add_two_ints`.

1.8.6 Strengths

Providing an API for fault injection and error emulation with a representative set of possible operations enables balck-box robustness testing. It also may save time for the testing team, which can build scenarios on top of the provided interface.

1.8.7 Weaknesses

Embedding an interface for enabling source code modifications or runtime disturbances can be time-consuming for developers whenever there are no third-party tools that provide the required operations.

1.9 14. Isolate components for testing

1.9.1 Context (WHEN)

Robotics applications require to be safely field tested with warranted no side effects on the running system or the environment, including personnel. To this end, it is fundamental to pursue isolation of the components being tested.

Man-in-the-Middle (MITM) can provide isolation of computing nodes for testing, which, as stated by Bertolino et al [4], can guarantee that the execution of the field tests does not interfere with the operation of the tested system. In addition, MITM nodes promotes the ability to drop or modifying internal signals improving the controllability of events and states, which as described by [8], refers to the degree to which a certain specification can be enforced on the system.

1.9.2 Reason (WHY)

The ROS ecosystem does not provide a native mechanism for node isolation. The MITM node strategy can be a powerful and flexible design pattern to isolate computing nodes in ROS-based systems. It allows for increased visibility into the system's operation, more flexible communication between nodes, and improved reliability and security.

1.9.3 Suggestion (WHAT)


Isolation of components is an important feature to enable field-based testing (by following the “let it crash” philosophy introduced by Netflix web:chaos-monkey) while avoiding the crash of the system. Isolation permits catching immediately the component that is crashing and executing countermeasures to keep the system up. Examples of solutions for isolation are: (i) the use of proxy nodes of ROSRV (git: ca2 suerdogan/ROSRV), or (ii) introducing intermediary nodes between original nodes by exploiting the topic remapping functionality (wiki: remap) of ROSMonitoring (git: autonomy5 and-verification-uol/ROSMonitoring), which enables swapping topic names just before running the application.

1.9.4 Process (HOW)

In ROS, the man-in-the-middle (MITM) strategy consists of introducing intermediary ROS nodes between components in this context the term ‘components’ stands for one or more ROS nodes presenting unique functionality and well-defined interfaces. Such intermediary nodes may react by blocking, dropping, modifying, or inspecting the messages passed between components. We identified two means of enacting MITM in ROS, through a proxy design pattern, e.g.,

ROSRV [14], or through topic remapping, e.g. ROSMonitoring [37]. The former works by generating a proxy-based of the ROS Master node which has full control of the nodes and topics during runtime, deploying an intermediary node for message diverting when required. The latter launches an intermediary node and performs topic remapping before execution – asking for the topic names that should be diverted.

1.9.5 Exemplars

Proxy. ROSRV [14] deploys a ROS master node proxy, namely RVMaster , that acts as a firewall, blocking the access to the ROS master node port. This node, in turn, is responsible for creating MITM nodes (also called monitors) that intercept the messages flowing between nodes in the system. RVMaster instantiates the MITM nodes according to user-provided specifications of access policies, even though it does not require any change to the system under scrutiny. Listing 4 portrays the `createInterceptors()` method from the RVMaster that enacts MITM monitor nodes by re-subscribing to a new set of topics.

Listing 4: Creating proxy-based MITM nodes in ROSRV.

```
void ServerManager::createInterceptors() {
    for (std::map<std::string, std::string>::iterator
         it=rv::monitor::topicsAndTypes.begin(); it!=rv
         ::monitor::topicsAndTypes.end(); ++it) {
        string topic = it->first;
        string datatype = it->second;
        Monitor *monitor_p = NULL;
        string monitorname = topic+MONITOR_POSTFIX;

        XmlRpc::XmlRpcValue params, result, payload;
        params[0] = monitorname;
        params[1] = topic;
        params[2] = datatype;
        string m_uri = "";
        params[3] = m_uri;


        //create a xmlrpcmanager-client for each
        monitor
        monitor_p = new Monitor(params, rv_ros_host,
                               monitorname);
        monitorMap[topic] = monitor_p;
        m_uri = monitor_p->getMonitorXmlRpcUri();
        params[3] = m_uri;

        bool f = master::execute("registerSubscriber",
                                 params, result, payload, true);

        //start a new thread with the params
        boost::thread(boost::bind(&Monitor::
                                   monitorThreadFunc, monitor_p));

        result[0]=1;
        result[1]="Subscribed to ["+topic+"]";
        XmlRpc::XmlRpcValue pubs_monitor;
        pubs_monitor[0] = m_uri;
        result[2] = pubs_monitor;

        ROS_INFO("Monitor %s successfully registered a
                  subscriber to topic %s", monitorname.c_str()
                  , topic.c_str());
    }
}
```

Topic remapping. ROSMonitoring  [37] utilizes topic remapping to ensure safety and security in the system by

creating intermediary nodes via topic name remapping (roslaunch/remap). Remapping is a core tool from ROS that enables swapping topic names just before running the application. *Remap*, enables inserting an intermediary node between the other two original nodes. Listing 5 exemplifies the topic remapping-based MITM strategy applied by ROS-Monitoring. The code snippet shows that the launch files are incremented with remap statements including the node names, original ('from') topic and new ('to') topic name.

Listing 5: Topic remapping-based MITM in ROSMonitoring.

```

1 def instrument_launch_files(nodes):
2     ...
3     for (name, package, topics) in launch_files[path]:
4
5         if node.get('name') == name and node.get('pkg
6             ') == package:
7             for topic in topics:
8                 remap = ET.SubElement(node, 'remap')
9                 remap.set('from', topic)
10                remap.set('to', topic + '_mon')
11            break

```

1.9.6 Strengths

The MITM strategy enables freezing of the application at the software level without the normal operation, and rollback in case the test scenarios are prone to side-effects and do not require any change in the code.

1.9.7 Weaknesses

Placing nodes between computations may cause performance overhead in the system under scrutiny. In addition, erroneous intermediary nodes may affect the functionality of the system, and thus represent a threat to the test's reliability. This guideline does not directly apply to server-based or parameter-based communication.

2 GUIDELINES FOR QUALITY ASSURANCE THROUGH RUNTIME VERIFICATION AND FIELD-BASED TESTING

In this section, we describe twelve guidelines to assist quality assurance teams to attaining confidence on robotic applications developed using the Robot Operating System (ROS). Attaining confidence, in this case, consists of applying runtime verification and field-based testing. The Prepare Execution Environment (PE) group contains two guidelines: PE1 to warn about the overhead acceptance criteria, and PE2 to create models for runtime assessment. The Specify (Un)-Desired Behavior group (SDB) contains three guidelines concerning the specification of desired and/or undesired behaviors. SDB1 and SDB2 concern the specification of properties through the use of logic-based languages (SDB1) or Domain Specific Languages (DSLs) (SDB2). SDB3, in turn, focuses on scenario-based specifications of test cases. The Generate Monitors and Test Cases group (MTA) contains two guidelines. MTA1 focuses on how to improve the robustness of the system by performing noise and fault injection. MTA2 discusses how to exploit automation for monitoring and testing, e.g., generation and prioritization of test cases. The System Execution group (SE) contains two

guidelines focusing on the importance of using record-and-replay when performing exploratory field tests (SE1) and the importance of headless simulation (without GUI) for optimization and/or automation (SE2). Finally, the Analysis & Reporting group (AR) contains two guidelines. AR1 focuses on performing postmortem analysis to diagnose non-passing test cases. While AR2 focuses on the use of reliable tooling to manage field data.

2.1 PE1. Understand the overhead acceptance criteria

2.1.1 Context (WHEN)

According to practitioners, performance is among the three most important quality attributes when designing a robotic application in ROS [10]. Performance is often important in robotics because many computations performed by robots tend to be data-intensive, e.g., computer vision, planning, and navigation [10]. Gaining confidence in robotic applications, then, should not interfere with the nominal performance of the robot under scrutiny. Less (or no) impact on performance is especially desirable when the assurance gathering process interacts with the running system, for instance, in-the-field testing and runtime verification. We name the extra load available for gaining confidence on ROS-based applications as *overhead acceptance criteria*. The overhead acceptance criteria can be allocated to monitoring, isolation, or maintaining the security of privacy during the runtime assessment session [4].


2.1.2 Reason (WHY)

A rule of thumb says that more observations tend to enable more precise assurance arguments within a limit. Observing, however, is never free from side effects, incurring in overhead [8]. Therefore, the quality assurance team must contrast the precision of the assurance arguments against overhead introduced by the observation medium. The overhead acceptance criteria are the basis for deciding the runtime assurance strategy.

2.1.3 Suggestion (WHAT)



The use of runtime verification or field-based techniques might add computation overhead. The QA team should understand how much overhead is acceptable; this is important to decide on a test strategy that will not severely impact the performance of the running system. Such overhead may be due to monitoring with `ros_tracing` (git: `ros2/ros2_tracing`), component isolation, or security and privacy maintenance overhead with `ROSploit` (git: `seanrivera/rosploit`).

2.1.4 Process (HOW)

Typically, understanding the overhead acceptance criteria follows from contrasting the required performance for delivering the required service, aka nominal performance (e.g., time to reaction, latency, speed), against available resources (e.g., computing power). The QA team may use off-the-shelf ROS tooling, like `ApexAI/Performance_test` , to understand the nominal performance of the ROS-based application. The difference between nominal performance and expected performance may be allocated to the overhead acceptance criteria. If the nominal performance is close enough to the expected performance, there is not enough

space for implementing runtime assurance techniques. We categorise three types of overhead that may affect the runtime assurance process: Runtime Monitoring [37], [44], Isolation overhead [45], [46], and Security and Privacy overhead [47], [48].

2.1.5 Exemplars

Monitoring overhead is all extra load put on the system-under-test due to gathering, interpreting, and elaborating data about the execution. For example, ROSMonitoring (autonomy-and-verification-uol/ROSMonitoring ) [37] determines the monitoring overhead by calculating the delay introduced in the message delivery time between ROS nodes. The authors analyse the overhead by varying the size of the system under monitoring, message passing frequency, and number of monitor nodes. However, their overhead analysis is not transparent with respect to gathering, interpreting or elaborating on data, the analysis looks at the monitoring overhead as a black box. Another example is `ros2_tracing` (`ros2/ros2_tracing` ) [44] that provides a tool for tracing ROS2 systems with low latency overhead. The authors measure monitoring overhead by collecting the time between publishing a message and when it is handled by the subscription callback. In short, monitoring and tracing tools add some overhead that typically affects the message passing latency, the QA team must understand and define a precise time allowance to this overhead.

Isolation overhead is all extra load put on the system-under-test for guaranteeing that the runtime assessment will not interfere with the normal operation or produce undesired side effects. As an example, Lahami M. et al. [45] propose a safe and resource-aware approach to test dynamic and distributed systems. Safe by employing testing isolation techniques such as BIT-based, tagging-based, aspect-based, cloning-based, and blocking-based. Resource aware by setting resource monitors such as processor load, main memory, and network bandwidth. The authors show that the overall overhead is relatively low and tolerable, mainly if dynamic adaptations are not commonly requested. However, there is no fine-grained evaluation of the overhead introduced by isolation techniques. In fact, isolation overhead is rarely reported [46].

Security and Privacy overhead is all extra load put on the system-under-test for maintaining security and privacy constraints while testing. For example, ROS-Immunity is a security tool for preventing ROS-based applications from malicious attackers. Rivera S. et al. [47], determines overhead for maintaining the ROS-based system secure, while operating, in terms of power consumption (in Watts) by comparing the power draw of the system with and without their tool. From another stance, Breiling B. et al [48] present a secure communication channel to enable communication between ROS nodes using protocols such as Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) per each ROS topic in the application. The protocols follow three steps: an initial handshake with mutual authentication, using symmetric encryption (AES-256), and using Message Authentication Codes (MAC) for data integrity. Importantly, they evaluate the overhead introduced for each step, totalling

in a few percentage points (2%-5%) of increase in average CPU load.

2.1.6 Strengths

Understanding the overhead acceptance criteria in advance may avoid unexpected interruptions in the ROS application functionality due to runtime quality assessment procedures. For example, when testing procedures are mistakenly scheduled whenever the ROS-based application is operating under a high load. Learning about the overhead in advance criteria may also ask for re-design due to a lack of verifiability during runtime. For example, when the nominal performance of the ROS system is close to the expected performance, in value.

2.1.7 Weaknesses

Although there is tooling to support the assessment of the overhead acceptance criteria, the process for understanding involves executing the system, collecting performance data and analysis. This may conflict with time-to-market requirements, asking for further negotiation with the business goals of the ROS-based system.

2.2 PE2. Create models for runtime assessment

2.2.1 Context (WHEN)

Gaining confidence in the running systems can be costly due to the need for isolation from side effects, preservation of security and safety, controllability, and observability [4]. One way to address this issue is through the use of model-based runtime assessment, which can help to manage complexity and ensure the safe operation of ROS-based applications [49]. Models can be used to define safety criteria, create digital twins of the system and its environment, and ensure adequate test coverage. This guideline discusses creating models for managing the complexity of runtime assessment in ROS.

2.2.2 Reason (WHY)

Modeling is essential for efficient and effective testing of ROS-based systems in the field. ROS-based systems lack a built-in system representation, making it difficult to identify and address bugs without costly calibration of sensors and actuators [50], [51], [6]. By incorporating models, such as those used in low-fidelity simulation, the testers can reveal the same bugs found in real-world navigation [52]. Implementing models in runtime assessment of ROS-based systems can save costs and improve the overall performance of the tests.



2.2.3 Suggestion (WHAT)

The QA team might create and exploit models of the system and/or of its environment (a sort of digital twin) for runtime assessment, predictive maintenance, checking alternatives, and so on. For example, they can create a digital twin of the system by using CPSAML (`me-big-tuwien-ac-at/cpsaml`), or formal tools such as UPPAAL combined with UPPAALTron (doi: 10.1109/ECMR.2015.7324210). In addition, the QA team may use ROS metamodels (`ipa-nhg/ros-model`) to facilitate the use of tools and graphical plug-ins for reverse engineering models from ROS code.


2.2.4 Process (HOW)

Models can be used to create useful representations of the system's components for runtime assessment of ROS-based applications, such representations include (i) models of the system itself, (ii) models of the environment and (iii) metamodels. (i) Models of the system itself consider digital twins of the system or part of the system [53], [54]. (ii) Models of the environment may be carried out in tools for formal verification, e.g., UPPAAL [55], [56]. (iii) Metamodels rely on the Eclipse Ecore technology to encoding rules for modeling, generating and introspecting ROS-based systems [57], [58]

2.2.5 Exemplars

Models of the system itself. Saavedra Sueldo C. et al. [53] proposes an architecture for integrating digital twins (DTs) on production systems using ROS. The paper argues that digital twins should be used as the central component for the formulation of autonomous decision-making systems due to their static nature. In order to do this, in their online repository (INTELYMEC/ROS_Tecnomatix ) the authors provide a ROS node (named `plant_simulation_node`) that performs the same operations (e.g., decision using a method of reinforcement learning and a material handling procedure) of its counter-part, but in a controlled environment. The simulation node can be understood as a mock or model of self. In a similar direction, Fend A. and Bork D. [54] suggest model-driven runtime monitoring based on digital twins for ROS-based applications, namely $CPSA_{ML}$ (me-big-tuwien-acat/cpsaml ). $CPSA_{ML}$ generates local DT components that are executed as ROS nodes. The DT component holds the state of the DT entity; in other words, it provides a model of the execution life-cycle of its counterpart. Ultimately, the model of the system can be used for reasoning during testing and runtime verification.

Models of the environment. The test setup described in Ernits et al. [55] uses UPPAAL Tron [56] as the primary test execution engine. First, the QA team should model, in UPPAAL, (i) an implementation model of the system under test and (ii) a topological map of the environment. Second, the QA team integrates an adapter, provided by Ernits et al. [55], which is responsible for translating messages between the environment model and the appropriate topics in ROS, acting as the interface between the ROS-based system under test and the UPPAAL Tron model of the environment. Finally, the interaction between the implementation model (i) and the environment model (ii) is monitored during system execution and afterwards, the equivalence between the measured outcomes from the running system and the outcomes from the model is checked. The operating environment, however, is typically prone to uncertainty which makes modeling the environment a costly activity.

Metamodels. One of the most notable works using models as a foundation for software development with ROS is the series on Bootstrapping MDE Development from ROS Manual Code [57], [58] by Hammoudeh Garcia N. In the repository (ipa-nhg/ros-model ) a set of metamodels defined as Ecore models are provided to facilitate the use of tools and graphical plug-ins for creating models from ROS code, composing and validating model compositions, auto

generating deployment artifacts, and checking the use of standard specifications. The repository also includes tutorials on reverse engineering from ROS code to models, creating a model using introspection at runtime, and generating ROS code from models.

2.2.6 Strengths

By using models for field-based testing, it is possible to reveal bugs in a safe and cost-effective way, without the need for costly and risky test campaigns in the field; It also allows for flexibly choosing the model representation, and the level of detail of the model, according to the needs of the system, the test scenarios, and the available resources. Models can be reused in different testing scenarios, and across different system components, reducing the effort and time required for testing.

2.2.7 Weaknesses

Large and complex robotics systems may impair the creation and maintenance of runtime models. Models need to be updated and maintained as the system evolves, which can be a costly and time-consuming task. In addition, modeling can be prone to errors and inaccuracies, which can affect the quality of the testing results. Models are domain dependent and may not be easily transferrable between domains such as healthcare domain, automotive, or aerial domains.

2.3 SDB1. Property specification using a logic-based language

2.3.1 Context (WHEN)

Robotics systems are complex, inherently hybrid systems that combine hardware and software components and whose behaviour is often dictated by close safety, legal, and ethical considerations [9]. Furthermore, given the increasingly frequent deployment of ROS-based systems in safety-critical environments coupled with their dependence on complex decision-making and sophisticated control systems, it becomes necessary to find a form of verification that can reliably assess the correctness of these systems. This guideline discusses the use of formal and logic-based languages for specifying properties for runtime verification and/or field-based testing.

2.3.2 Reason (WHY)

A logic-based language for verification and/or testing allows specifying properties that describe observable actions, outputs, how they relate to each other and when they should manifest [59]. The behaviour is often described using temporal logic, which enables the specification of desired or undesired interactions amongst multiple components (ROS nodes) regardless of the complexity of the system. To simplify the specification of properties in temporal logic, some of the tools refer to existent property specification patterns [60], i.e., a collection of existing known recurrent patterns for the specification of temporal properties. Specification patterns also enable the properties formulation in English thanks to a bidirectional mapping from various temporal logics to a structured English grammar.

2.3.3 Suggestion (WHAT)

The QA team should be prepared to specify properties using unambiguous and precise languages like logic-based languages, as often required by verification tools. User-friendly instruments, like specification patterns <http://ps-patterns.wikidot.com/>, might facilitate the error-prone specification process and make the specification accessible to people lacking expertise in logic. For example, HAROS ([git: git-afsantos/haros](https://github.com/git-afsantos/haros)) uses a logic-based language called HPL for property specification that is used to synthesize runtime monitors for testing and verification.

2.3.4 Process (HOW)

There are several choices of tools that use logic-based languages for properties specification [61], [62], [13], [63], [64]. These tools allow for safety, security, and liveness analyses. The formal specification languages for ROS-based systems should provide references to individual resources (e.g. topics) and message contents as well as temporal operators and relations in addition to real-time behaviour specifications.

2.3.5 Exemplars

The identified tools make use of various types of temporal logic.



Linear Temporal Logic (LTL). The work in [64] proposes a novel architecture, for assertion-based verification by using monitors synthesized automatically from Linear Temporal Logic (LTL) assertions. Then, the monitors are encapsulated into plug-and-play ROS nodes and docker containerization is used to improve system portability. As an example, Listing 6 shows an LTL assertion to check if a robot arrives at a position $\langle x_2, y_2 \rangle$ before a certain timeout.

Listing 6: LTL assertion to check if a robot reaches a desired position

```


1 always((robot x1 = x1 && robot y1 = y1 && newGoal)
2 implies
3 (currentTime < timeOut ∨ robot x2 = x2 && robot y2 =
  y2))

```

Signal Temporal Logic (STL). Signal-temporal logic is a special case of MTL where properties are defined over signals. They enable real-time reasoning and constraint specification. In [63], [65], Signal Temporal Logic (STL) is used to describe Cyber-Physical Systems (CPS) properties. The approach introduces the novel quality of *robustness semantics*, which implies the system's ability to measure how far is an observed behaviour from satisfying or violating a specification. Concretely, the developed tool was **rmtat** , which was later augmented to support integration with ROS-based systems with **rtamt4ros** .

Metric Temporal Logic (MTL). Some runtime verification tools add the ability to handle real-time specifications since the reaction time can have a strong influence on the success or failure of the robot's mission. Taking this aspect into consideration, the RV tools described in [61], [13] use Metric Temporal Logic. On the other hand, [62] addresses it by combining Past-Time LTL to express complex formulas with timed constraints on the evaluation of these formulas. Hu et al. design their robot monitoring specification language based on Discrete Time-MTL (DT-MTL)s [13]. The language's

syntax embeds a time interval in the traditional MTL notation. Their approach uses the system's periodic nature to discretize the real-time property and replace the real number in time constraints with the number of CPU clock cycles.

TCTL. Timed automata is one of the most widely used formal models to specify and verify real-time systems. Rodrigues A. et al. specify timing constraints using Timed Computational Tree Logic (TCTL) to encoding the requirements of a safety-critical healthcare monitoring system in ROSseams18/uppaal  [66]. Halder et al. specify UPPAAL models over ROS applications by performing manual code analysis. This phase requires the extraction of ROS code parameters that affect the desired properties of ROS-based robotic applications, including the publishing rate, the subscriber's spin rate to process callbacks, the time to transmit messages over channels, and the time to process callbacks. Properties are specified over queue overflow constraints [30].

Patterns-based specification. In [61], the authors introduce HAROS, which is a framework for quality assurance of ROS-based code. The use of the formal language HPL ensures that after entering a state of high priority (where the messages in the topic *'state'* have *data* greater than 0), the system should remain in that state for, at least, one second. This can be accomplished through the following property definition: **"globally: /state {data > 0} forbids /state {not data > 0} within 1000 ms"**. As can be seen, the property specification makes use of specification patterns, similar to the *absence* pattern with a time constraint [60]. Recently, other sets of patterns specific to robotics have been defined [67], [68]; even though they have been conceived for mission specification, they could be profitably used for verification and (scenario-based) testing.

2.3.6 Strengths

Using logic-based languages to specifying properties offers a standardized approach to validation in compliance with well-adopted specification pattern.

2.3.7 Weaknesses

It is noted that the properties' expressiveness is often limited by the description language meaning that different languages and tool sets may enable specific property specifications or not.

2.4 SDB2. Use Domain Specific Languages to Specify Properties

2.4.1 Context (WHEN)

To address the safety challenge, one option is defining a clearly specified and isolated layer, which is declared separately from the main program and that can express the functional safety-critical concerns in terms of externally observable properties of the software [69]. This guideline discusses the use of a Domain Specific Language to define the properties of the system.

2.4.2 Reason (WHY)

As a more informal, code-like alternative, the QA team has the choice of utilizing a Runtime Verification tool that has a built-in ROS-tailored language and allows the quality assurance team to specify and validate the correct behaviour of the system.

2.4.3 Suggestion (WHAT)

In complement to logic-based instruments, the QA team may opt to use verification tools that allow code-like specifications of properties to simplify the definition of the desired behavior. For example, ROSMonitoring (git: `autonomy-and-verification-uol/ROSMonitoring`) allows for code-like specifications of properties in a domain specific language (DSL) targeted to the properties such as writing assertions over the robot's position using if-else constructs.

2.4.4 Process (HOW)

The works in [69], [37], [14] describe DSLs for specifying properties. The properties described with the DSL are checked mainly by intercepting the messages from relevant services and topics from the ROS system. It is important to consider the nature of the properties that should be specified and the expressiveness of the DSL. These DSLs give often the possibility to specify actions that should be performed in reaction to a violation of a safety property. It is important that the QA team chooses, customizes or conceives the DSL that best suits the specific needs.

2.4.5 Exemplars

The runtime verification tools described in [37], [14] are based on a DSL that allows collecting the information from the different software components (ROS nodes) at a given moment in time and perform a corresponding action if one or more of the predefined rules are broken. In [14], the authors define ROSRV. Listing 7 shows the definition of a condition for which the system will be monitoring and the corresponding action to be taken in case a violation occurs.

Listing 7: Example of position and velocity property specification in ROSRV

```

1 event moveOrStop(double lx, double ly, double lz)
2 /landshark_control/base_velocity
3 geometry_msgs/TwistStamped
4 '{twist:{linear:{x:lx,y:ly,z:lz}}}'
5 {
6   if( posx > 9 ) {
7     if(vectorx * lx < 0) {
8       ROS_WARN("Position not allowed");
9       return;
10    }
11  }
12 }
```

In ROSRV, the event definition structure is as follows. First, the keyword event must be followed by the event name and the parameters that need to be provided to the method. Consequently, one must state the topic name, data type and the mapping from the topic information to the method parameters. Such structure is shown in Listing 7.

In [69], the property definition DSL (namely RuBaSS) allows incorporating time into the rule definition. For example, one can describe a safety property as `"max_speed_exceeded: linear_speed > max_speed for 2 sec;"` where the `linear_speed` and `max_speed` values are extracted from messages published in corresponding topics. Consequently, this property checks whether the comparison holds true for 2 seconds using the DSL defined by the authors.

2.4.6 Strengths

The use of Domain Specific Languages for specifying properties brings a programmer-friendly interface for runtime validation. Moreover, it promotes the definition of well-structured and standardized approaches.

2.4.7 Weaknesses

On the negative side, the expressiveness of properties is limited by the DSL used.

2.5 SDB3. Use languages and tools to scenario-based specification of test cases

2.5.1 Context (WHEN)

Scenario specification allows the QA team to test robots on credible and complex activities since it defines the course of a robotic mission [16]. Gathering confidence in autonomous systems through so-called verified scenarios is a cornerstone of simulation testing [9]. However, designing test cases in the form of scenarios implicitly requires accounting for possible variations and conditions. This makes scenario-based testing a challenging task and often deemed infeasible. In addition, unexpected scenarios frequently underlie failures in robotic systems [16]. Therefore, it is important to consider real-world data during scenario generation since it permits to focus on a range of scenarios that can be unexpected during the design phase [4].

2.5.2 Reason (WHY)

Scenario-based test case generation is a powerful approach to verify ROS-based applications because it allows for the systematic exploration of different situations and conditions that the robotic system may encounter in the real world. By defining scenarios, the testing team can ensure that the application behaves correctly under a wide range of circumstances, including both expected and unexpected events.

2.5.3 Suggestion (WHAT)

The QA team might integrate scenario specification languages and tools to enable a systematic exploration of real-world situations and conditions for ROS-based applications. For example, Geoscenario (git: `rodrigoqueiroz/geoscenario-server`) uses behavior trees to program dynamic interactions between the system-under-test and other vehicles in the scenario. In addition, SCENIC (git: `BerkeleyLearnVerify/Scenic`) is a probability-based programming language that enables the specification of rare events in environment models that are used to generate test cases for vehicles running on the CARLA simulator (git: `carla-simulator/carla`) that may be further integrated to testing ROS-based systems.

2.5.4 Process (HOW)

To implement scenario-based specifications for testing ROS-based applications we recommend describing technology enablers and motivating use cases. Technology enablers, in this case, list a set of languages that assist the design of scenarios for testing ROS-based applications. The technology-enablers vary with respect to human-vehicle behavioral

description [70], to probability-based programming of scenarios [71], and integrating graphical simulation with Simulink-based control design and ROS [72]. From the point-of-view of use cases, scenario specification may turn useful to perception monitoring [73], validating multi-layered control strategies [74], and pedestrian simulation scenarios [75].

2.5.5 Exemplars

Technology-enablers: GeoScenario [70] proposes using a behavior tree-based DSL to describe autonomous human-vehicle models for scenario-based testing. The toolset advocates for scenario specifications that reflect dynamic interactions between humans and the subject system in real traffic conditions. Therefore, behavior trees are used as a fundamental control-flow strategy. GeoScenario interfaces with a simulation layer where LIDAR and camera capture data from the vehicle controller implemented at ROS-level. Listing 8 showcases an implementation of one of the agent’s behavior in a pre-crash scenario from NHTSA. The vehicle changes the lane to the left, then continues driving using another drive_tree implementation. The operator ‘?’ stands for fallback operation, ‘-’ is sequential, ‘condition’ evaluates to a Boolean value, and ‘maneuver’ executes a task.

Listing 8: Lane change human-vehicle implementation in Geosenario.

```

1 behaviortree LaneChange:
2   ?
3   ->
4     condition c_trigger(sim_time (tmin=4))
5   ->
6     condition c_reach(gap(target_lane=Left, range=38,
7       repeat=False))
8     maneuver cutin(MCutInConfig(target_lane=Left,
9       delta_s=(5, -3, 0)))
10    subtree drive_tree(m_vel_keep=MVelKeepConfig(vel
11      MP(14.0, 10, 6)))

```

SCENIC [71], a probability-based programming language, argues for the specification of rare events in environment models for autonomous vehicles and robots for testing purposes. The language relies on the specification of *scenarios*, i.e., configurations of physical objects and agents, and how they change over time. In combination with CARLA [76], SCENIC can be used to specify scenarios for ROS-based simulation test scenarios. For example, departing from a Metric Temporal Logic (MTL) safety specification, SCENIC generates (with VERIFAI [77]) 2000 scenarios for a vehicle performing a right turn at an intersection, yielding to the crossing traffic.

Supporting Software-In-the-Loop (SIL) simulation testing, Xu C., et al. [72], introduces a platform that unites PreScan [78], MATLAB/Simulink [79], and ROS. The three-layered platform enables virtual sensors and scenarios designed in PreScan, control design in MATLAB/Simulink, and environment perception, planning and control execution at ROS-level. The proposed stack features graphical scenario specification including sensor simulation and may be used for mobile robotics with a focus on self-driving simulation testing.

Use Cases: The work in [73] integrates PerceMon [80], a tool

for monitoring spatio-temporal specifications for perception systems, with the CARLA simulator (integrated with ROS), and, in cohort with the OpenSCENARIO specification format defines a set of high-level scenarios for their experimentation setting. Their experimental scenario features sunlight exposure, cyclists crossing or poorly occluded pedestrians, and rendering corner case test fixtures. Similarly, [74] utilizes a ROS-CoppeliaSim [80] integration to validate their three-layered control decoupling strategy for lateral and longitudinal motion. The tool permits defining motion constraints to the mechanical joint points of the high-precision vehicle model, which forms a front-wheel steering and rear-wheel drive smart car model powered by Ackerman steering. [75] uses *pedsim_ros* [81], a pedestrian simulator tool that relies on XML scene specification, for example Listing 9.

Listing 9: Corridor scenario specification with *pedsim_ros* (Link to source).

```

<?xml version="1.0" encoding="UTF-8"?>
<scenario>
  <!--Obstacles-->
  <obstacle x1="-0.5" y1="-0.5" x2="29.5" y2="-0.5"/>
  <obstacle x1="-0.5" y1="-0.5" x2="-0.5" y2="14.5"/>
  <obstacle x1="-0.5" y1="14.5" x2="29.5" y2="14.5"/>
  <obstacle x1="29.5" y1="-0.5" x2="29.5" y2="14.5"/>

  <waypoint id="east" x="5" y="5" r="2" b="1"/>
  <!-- Sink -->
  <waypoint id="west" x="25" y="5" r="2" b="2"/>

  <waypoint id="robot_goal" x="22" y="27" r="2"/>
  <waypoint id="robot_start" x="4" y="4" r="2"/>

  <agent x="4" y="4" n="1" dx="0" dy="0" type="2">
    <addwaypoint id="robot_start"/>
    <addwaypoint id="robot_goal"/>
  </agent>

  <!--AgentClusters-->
  <source x="2" y="5" n="8" dx="3" dy="5" type="0">
    <addwaypoint id="east"/>
    <addwaypoint id="west"/>
  </source>
</scenario>

```

2.5.6 Strengths

Scenario-based testing provides a systematic and comprehensive approach to verify ROS-based applications under various conditions. The integration of scenarios with ROS-compatible simulation environments allows for realistic and repeatable testing.

2.5.7 Weaknesses

The exemplars provided are primarily focused on the automotive domain, which may limit the generalizability of the guideline to other robotic application domains.

2.6 MTA1. Improve the robustness of the system by performing noise and fault injection

2.6.1 Context (WHEN)

When testing a robotic system, the QA team needs to gain confidence that it will behave safely when faced with unexpected inputs [27]. Techniques such as Fault Injection (FI) and Noise Injection (NI) can aid the QA team in identifying

weaknesses, evaluating the resilience of the system as well as measuring its performance under stress. The deliberate introduction of faults such as hardware failures, network interruptions, software errors and signal alterations, can help assess how well the system recovers from these issues. In addition, fault injection is the central tool of a technique called Fault Based Testing, in which the generated test cases address potential faults that can be foreseen at design time[4].

2.6.2 Reason (WHY)

Testing techniques based on fault injection and noise injection aid in increasing the overall reliability of the system under testing.

2.6.3 Suggestion (WHAT)

The QA team can use tools that generate noise or inject faults to gain confidence that a robotic system will behave safely when faced with unexpected situations. For instance, RoboFuzz (git: sslab-gatech/RoboFuzz) enables the generation of faults (in ROS 2 applications) through message mutation with three intents: violation of physical laws, violation of user-specified properties, and cyber-physical discrepancies. Moreover, RosPenTo (git: jr-robotics/ROSPenTo) enables security assessment by (un-) registering publishers or subscribers, isolating nodes or services, and injecting false data in the messages in ROS 1 applications.

2.6.4 Process (HOW)

Fault injection is a testing technique to evaluate the robustness of a system that consists of intentionally introducing faults, failures and errors into the software system. The fault injection can occur at various levels, these generally are defined as the software, the hardware and the network layer. In the software layer, for instance, typical operations include the modification of variables and the manipulation of input data. Noise injection, instead, refers to introducing noise or interference into a system with the aim of improving the system's resilience. Several tools have been developed in recent years to apply these concepts in the ROS domain, either based on noise injection `ros1_fuzzer` [81] or `ros2_fuzzer` [81] or fault injection [41], [82] *ROS Fault Injection Toolkit* `camfitool`.

2.6.5 Exemplars

Noise Injection: With the goal of effectively stressing a data-driven ROS system, Seulbae Kim et al. [81] implemented RoboFuzz, which is based on a data type-aware mutation technique aimed at finding correctness bugs in the system. RoboFuzz takes a target system and a test strategy as input and outputs the report of found bugs after performing a fuzzing technique based on message mutation. In addition, RosPenTo [83] (jr-robotics/ROSPenTo) provides the operations of unregistering and registering publishers/subscribers, isolating nodes and services, and injecting false data in messages. As an example, the authors show how to use the tool to isolate the safety monitor node and to inject fault data in a robotic operation in such a way that the robot may harm humans [83].

Fault Injection: Targeting the inclusion of Fault Injection in the system, the `imfit` tool, which is based on applying

mutation testing to relevant files with the ROS source code, is of great value [41]. The tool allows the user to create fault injection plans and select the operating conditions for the mutation process. Furthermore, fault metrics and diagrams can be obtained afterwards to analyze the system's response.

In addition, ROS Fault Injection Toolkit is a tool that aims to test the reliability and fault-tolerance of a ROS-based system while allowing for user-controlled fault injection, targeting especially the domain of image processing for autonomous driving. Similarly, `camfitool` is an official ROS tool which also allows the user to inject faults in images, but it additionally provides a simple intuitive graphical interface as well as the ability to perform the injection retrospectively or in real-time.

In the domain of Unmanned Aerial Vehicles (UAV), the authors in [82] present a ROS-based application the system is built as a ROS node and leverages the ROS communication protocol and Linux system to inject faults. As an illustrating example, the authors analyse the effect of corrupting the execution of modules related to the generation of the flight command and measure the impact on the quality of flight.

2.6.6 Strengths

It improves the reliability of the system. The faults injected can mirror real-world conditions. Demonstrating resilience can build confidence in the system.

2.6.7 Weaknesses

Designing and implementing fault and(or) noise injection scenarios can be complex. Injected faults might lead to scenarios that never occur in the real world. The injection process can be resource-expensive and time-consuming.

2.7 MTA2. Exploit automation for test case generation, test case prioritization and selection, oracle and monitor generation

2.7.1 Context (WHEN)

Automation of testing activities is getting increasing attention in the robotic domain, even in the case of field-based testing. Automation can be exploited for various activities, including test case generation [27], prioritization, selection, and oracle generation. The benefits brought by automation are various: it helps improving the efficiency, effectiveness, and reliability of the software testing processes, e.g. by increasing the test coverage and providing consistency and repeatability in the testing process. However, when applying automation in testing there are several challenges to be considered [16]. First, it needs to deal with the environmental complexity and its high variability. Second, specifying and/or generating an oracle that automatically distinguishes between correct and incorrect behaviours is not always obvious. Consequently, the test automation in field-based testing, is still limited and relies heavily on human contributions [4].

2.7.2 Reason (WHY)

Automating testing activities can improve efficiency, as they can be run quickly and repeatedly without human intervention. This leads to quicker feedback on the quality of the software. Also, automation is essential to enhance the efficiency, effectiveness and reliability of the software

testing processes. Moreover, as the system grows and evolves, automated test suites can generally be extended easily to accommodate new features and changes. Indeed, there is an initial investment to be made in creating and maintaining the automation machinery.

2.7.3 Suggestion (WHAT)

The QA team should exploit automation tools for test case generation, test case selection and oracle generation, as well as other testing activities, to efficiently gain confidence in ROS-based systems in the field. For example, Mithra (pdf: <https://afsafzal.github.io/materials/AfzalMithra.pdf>) learns oracles from logs generated during the execution, it is motivated by a case from ArduPilot and tested in autonomous racing cars built on ROS. In addition, HAROS (git: [git: git-afsantos/haros](https://github.com/afsanetos/haros)) promotes test case generation from properties using a tool called Hypothesis (git: [HypothesisWorks/hypothesis](https://github.com/HypothesisWorks/hypothesis)). Finally, a technical report (pdf: <https://www.cse.chalmers.se/~bergert/paper/2022-iroos-roboticstesting.pdf>) details how a company building mobile robots for disinfection uses equivalence partitioning for test case selection for the field.

2.7.4 Process (HOW)


To effectively implement test automation, one important step concerns the oracle specification, which is a mechanism or criteria to determine whether a test has passed or failed [84]. A novel approach in automatic oracle generation based on telemetry data which is validated using a ROS-based system is proposed in [85]. An important automation activity is the test case generation, with the aim of crafting tests cases to cover various functionalities, scenarios and edge cases of the system. For achieving this, a viable approach is to use Property Based Testing [86], where test cases are automatically generated to target specific properties of the system. Another important automation activity concerns the test selection, which aims to select, in the search for efficiency, the relevant and representative scenarios to be tested. In this area, equivalence testing is a technique that allows us to reduce the amount of test cases used by dividing the generated cases in equivalence classes, as evidenced in [87], where the authors study ROS-based systems in the domain of service robotics.

2.7.5 Exemplars

Technology enablers: As a technology enabler, `roctest`⁷ is a ROS framework that allows us to do full integration testing across multiple nodes. In the task of test case generation in particular, one crucial component of the library is unit test. This module makes it possible to perform ROS node integration-level tests as well as code-level unit tests. In the case of ROS 2, the testing mechanism is ingrained in the structure of the packages, which makes the test development considerably easier.

Oracle specification: In the area of automated oracle generation, the authors of [85] present *Mithra*, which is a novel and unsupervised oracle learning technique for Cyber-Physical Systems (CPS). The approach is applied to

autonomous racing cars built for ROS. The oracle learning approach builds oracles by clustering telemetry logs represented by a novel formulation of multivariate time series (MTS) that effectively and concisely encodes correct CPS behaviour.

Test case generation: Santos et al. [86] rely on property-based testing (PBT) to generate test cases driven by properties specified in the form of assertions. The tool Hypothesis  receives a model of ROS configurations and generates customizable property-based scripts for tests aimed at these configurations. It builds on top of configuration models extracted using HAROS, a static analysis framework for ROS applications.

As another example, in the domain of mobile robots, the work in [88] presents an approach to automatically generate test cases in the form of stressful trajectories. In essence, the work integrates kinematic and dynamic physical models of the robot to generate valid trajectories. The test case generator incorporates a parameterizable scoring model to efficiently generate valid yet stressful trajectories for a broad range of ROS-based mobile robots.

Test case selection: As a test selection mechanism, one commonly used technique is Equivalent Partitioning. The objective of equivalence partitioning is to minimize test case explosion, i.e. reduce the high number of possible combinations of the properties that will be tested. The idea behind Equivalent Partitioning is to group the test cases into equivalence classes according to how they handle valid or invalid data [87]. The authors report the experience of field testing in the domain of service robotics. In particular, the system under study was the Kelo AD disinfection robot whose software stack is based on ROS. To implement the equivalence testing, the criteria for defining the classes included robot motion, person posture, existence of occlusions and obstacle positions.

2.7.6 Strengths

Automation in testing activities can be executed repeatedly with low effort and cost. Moreover, it brings benefits in terms of efficiency, effectiveness, and reliability.

2.7.7 Weaknesses

There is a high initial investment of time and resources. There is also a maintenance overhead related to maintaining the automation machinery when the system evolves. Test generation may not scale for complex scenarios.

2.8 SE1. Use record-and-playback when performing exploratory field tests

2.8.1 Context (WHEN)

In order to effectively select which test scenarios should be escalated to the field, the testing teams often perform exploratory field tests [4]. Exploratory testing is an experience-based technique where the tester designs and executes tests based on prior knowledge, previous tests, curiosity, and other heuristics for common failures [89]. For instance, exploratory field tests may be useful for identifying corner cases that could be missed during design [87], or calibrating (e.g., tuning parameters [90], [91]) the system under scrutiny before committing to the field test scenario. ROS

7. <http://wiki.ros.org/roctest>

practitioners typically use Record and Playback (aka record-and-replay) for testing, debugging and developing new algorithms [16]. Record-and-playback is especially useful for performing exploratory tests of feedback loop systems since they provide lightweight means to record execution traces without preventing the testing team from freely exploring scenarios in the field. In ROS, record-and-playback is a technique that consists of recording message exchange between ROS nodes and enabling the user to reproduce the set of recorded messages in a specific order. The nominal use of record-and-playback, however, does not necessarily consider information from the field, placing the assessment (i.e., testing, debugging, analysis) in jeopardy.

2.8.2 Reason (WHY)

Escalating exploration scenarios to the field asks for lightweight means to recording data for scenario reproduction.

2.8.3 Suggestion (WHAT)

When running exploratory field testing, the QA team should use record-and-playback in order to keep track of the explored field scenarios, simplify error analysis, find and reproduce corner cases, and help with parameter tuning. The standard tool for record-and-playback in ROS is rosbag (wiki: <http://wiki.ros.org/rosbag>) but there are a few tool derivations supporting effective record-and-playback. For example, Rerun.io (git: [rerun-io/rerun](https://github.com/rerun-io/rerun)) promotes a graphical interface with a focus on the visualization of bag data leveraging common datatypes used on perception algorithms. In addition, NuBots (git: [NuBots/NuBots](https://github.com/NuBots/NuBots)) uses a genetic algorithm for tuning parameters for the RoboCup over data collected in field explorations after data bags.

2.8.4 Process (HOW)

Record-and-playback is well known by the ROS community. The ROS environment provides rosbag, a package that natively enables record-and-replay, which is frequently maintained and active. A few tools extend rosbags with addons to more expressive visualizations and data types (e.g., Rerun.io) or building on largely used behavior models (i.e., behavior trees) to replay robot behavior within mixed-reality scenarios [92]. Moreover, record-and-replay for exploring field testing scenarios used for simplifying error analysis [2], finding corner cases [87], and parameter tuning [91], [90].

2.8.5 Exemplars

Technology-enabler: The standard library for record-and-playback in ROS is rosbag (🔗) (or rosbag2 (🔗)). The rosbag package provides a means for recording exchanged messages in so-called bag files and reproducing the messages between components in release order. In addition, the package contains tools for analyzing, processing logs, and visualizing exchanged messages. Recent research, builds on top of rosbag to encode more expressive tooling for applying records and reproducing for testing purposes, for instance using behavior trees to replay robot behavior in mixed reality [92]. Rerun.io (🔗) provides a general-purpose tool with focus on visualization of bag data which leverages

common datatypes used in perception stack for robotics applications. Rerun enables URDF scene plotting, with 2d and 3d transformations, camera, odometry and tf transforms (e.g., [rerun/ros_node](#) example (🔗))

Simplifying error analysis: Beul M. et al. [2] extensively used bags to record the state of their unmanned aerial vehicle during field testing, keeping rosbag activated even in between experiments to avoid setting up overhead. Once the experiments are done, the authors filter and analyse the data using in-house tools including recording mission executions, editing traces and loading them for testing purposes. Finally, the authors claim that the toolset helped by simplifying the error analysis that was running over exploratory field scenarios (named experiments) where they logged user-defined missions in terms of ordered sets of 4D waypoints.

Finding corner cases: Ortega A. et al. [87] studied field testing experiences with an industrial-strength service robot (i.e., for hospital disinfection) transitioning from lab experiments to an operational environment. They conclude that the field testing strategies employed can be categorized into two phases, exploratory field testing and field endurance tests. Exploratory field testing helps find defects caused by variability in the environment, thus identifying corner cases [87]. However, replicating corner cases is challenging due to a lack of tooling to extract data from the field experiments. After all, the data collected from the fields must be (re-)useable, asking for tooling such as record-and-replay.

Parameter tuning: Algorithms using multi-objective optimization for improving functionality [91] or managing quality attributes (e.g., reliability) at runtime [90] may benefit from recording data in exploration scenarios and playback for tuning parameters. Zahn b. et al. [91] implements NuBots and uses a genetic algorithm (namely NSGA-II) to tune the parameters of a kicking strategy for a football robot player in the Robocup, for that, they rely on record-and-playback. NuBots (🔗), however, instead of relying on standard ROS tooling to record-and-playback provide their own implementation. Similarly, Caldas et. al [90] uses an implementation of their own to record whether a healthcare system (namely BSN (🔗) [23]) presents faults at task-level execution, then playback the recorded data in a data mining pipeline (using also NSGA-II) to tuning the parameters of a software controller.

2.8.6 Strengths

Record-and-playback is a good instrument to reduce the need for maintenance during quality assurance. It fosters also the repeatability of scenarios that would be rather lost when exploring field scenarios in an ad hoc way.

2.8.7 Weaknesses

Record-and-replay may not scale well [93], can be the door for security exploits [11], and lacks means for fine-grained control of their recording thus the emergence and need for extensions to rosbag.

2.9 SE2. No GUIs! Prioritize headless simulation

2.9.1 Context (WHEN)

Simulation is a great way to validate that a robot behaves as expected [16]. It can also be used in hybrid approaches

to combine real data from field with simulation engines, therefore making the testing more realistic [94]. Then, simulation can be useful as an isolation mechanism for testing, given that it can separate in-vivo testing from the production processes by reproducing the main process in a simulator based on information gather from the main execution by means of probes [4]. Furthermore, as mentioned in [27], the QA team can use replay and subsystem testing to check the behaviour of certain components of the system without a full simulation by starting them up in isolation and replaying the logs. Running the simulators stripped of their Graphical User Interface (GUI), also referred to as headless simulation, allows QA teams to (i) scale up the testing of the robot behavior towards large-scale experimentation, (ii) reduce the resources consumption, and (iii) properly organize experiment with automatic generation of scenarios, operation environments where the robots should act, and/or automatic injection of noise or uncertainty.

2.9.2 Reason (WHY)

Running the simulation headlessly allows one to avoid the overhead of rendering graphics and updating the GUI, as well as help in saving computational resources. Furthermore, headless simulation enables automation and make it feasible to integrate simulation with testing activities and incorporate it into continuous integration / continuous deployment (CI/CD) pipelines.

2.9.3 Suggestion (WHAT)

When it is possible to test or verify without human supervision, the QA team should prioritize headless simulation to avoid unnecessary overhead, enable large-scale experimentation, and facilitate integration with CI/CD pipelines. Example of tools supporting headless simulation are Gazebo (git: gazebo/gz-sim), V-REP (wiki: http://wiki.ros.org/vrep_ros_bridge), ARGOS (web: <https://www.argos-sim.info/>), MORSE (<https://morse-simulator.github.io/>), and MVSIM (git: MRP-T/mvsim). As a complement, OpenDaVINCI (git: [se-research/OpenDaVINCI](https://github.com/se-research/OpenDaVINCI)) interfaces with ROS and has been widely used for testing autonomous driving systems.



2.9.4 Process (HOW)


To run a simulator in headless mode, the user typically needs to specify a particular command-line argument or configuration setting before initiating the simulation. Although, in most of these simulators, the visual interface and the simulation engine are tightly coupled [95], many of them, including Gazebo, V-REP, ArGoS [96], [97], MORSE, MVSIM, and OpenDaVinci [98] allow for this setting to be set.

2.9.5 Exemplars

Technology Enablers: Most of the widely used simulators in the domain of robotics have a way to specify the intention of running the simulation headless. In Gazebo, it is done by specifying an option either through the command line or in the launch file⁸, although as stated in [95], there is still some instability in the support of this feature. V-REP also

allows a user to set a headless simulation by adding a special option in the execution of the program.⁹ Furthermore, the ARGOS simulator [96], created to support the simulation of large scales swarms of robots, efficiently supports a headless mode of operation; it has been proven to outperform Gazebo and V-Rep simulators when running headless in small scenes with up to 10 robots [97].

Other simulators that allow headless operations include MORSE , although the feature has only been tested in Linux, and MVSIM , which is a lightweight simulator capable of running real-time scenarios for multiple vehicles.

Use cases: The authors of [95] propose a paradigm to incorporate validation via headless simulation into the continuous integration cycle. In addition to providing an illustrating use case of how to build and connect the different component of such a system, they provide a set of generic steps to be followed during the setup phase. The authors of [98] develop a system in which they setup a virtual test environment targeting algorithms that will run in complex autonomous driving systems. As a result of this work, they build the open source environment OpenDaVinci, which includes libraries that enable the reuse of the algorithms to be tested in headless simulations as part of unit tests. They demonstrate how to use the framework in combination with the simulation environment to develop a self-parking algorithm .

2.9.6 Strengths

With headless simulation, the feedback is obtained faster, the simulations therefore can scale across multiple processors and cloud servers, and it makes possible to integrate the simulation into continuous integration pipelines.

2.9.7 Weaknesses

Debugging is more challenging without visual feedback as it is considerably more difficult to locate the issues or understand unusual behavior.

2.10 AR1. Perform postmortem analysis to diagnose non-passing test cases

2.10.1 Context (WHEN)

The test case execution phase results in reports yielding a set of ran test cases paired with Boolean results attesting whether the cases passed or failed. Failing test cases, often, indicate the reason for failure by comparing the expected outcome with the actual outcome. However, that may not be enough to understand *why did the robot fail* that test case, since such assertions do not comprehensively entail structural information about how the ROS-based system under test works. Such lack of information poses a risk to the usefulness of the executed tests. In order to provide fruitful information back to the development team, the testers should make the most out of the data by delving into a postmortem analysis for deriving explanations for the failed case [8].

2.10.2 Reason (WHY)

The postmortem analysis is essential for understanding the reasons behind the failure of test cases in ROS-based systems.

8. <https://answers.gazebo.org/>

9. <http://www.forum.coppeliarobotics.com>

2.10.3 Suggestion (WHAT)


The QA team should perform postmortem analysis and diagnose of non-passing test cases to explain the failures to developers or refine the arguments and confidence in the robotic system. For example, ROS projects may use a combination of Nagios (web: <https://www.nagios.org/>) and `ros/diagnostics` for monitoring, collecting and aggregating runtime data to diagnose failures. Moreover, CARE (git: `softsys4ai/care`) may be used for semi-automatic diagnosis of launch file misconfigurations or may rely on approaches such as Rason (git: `lsa-pucrs/rason/`) for multi-robot diagnosis.

2.10.4 Process (HOW)


Natively, ROS provides a diagnostics infra, the `ros/diagnostics` package¹⁰, the package enables manual data aggregation and analysis¹¹. However, unveiling explanations for the test case failure requires better support. The testing team may rely on visualization techniques [99], semi-automatic diagnostic generators [100], [101], and deployment of extra infrastructure to support explanations with more data [102].

2.10.5 Exemplars

Visualization: Often, roboticists need to use their technical experience to identify points of failure and diagnose failed test cases. With that in mind, Roman F. et al. [99] propose Overseer, an architecture that unites Nagios (i.e., a general-purpose open-source monitoring tool¹²) and `ros/diagnostics` (i.e., standard tooling for collecting and aggregating runtime data in ROS) as a key-enabler to visualization the runtime information and, in consequence, diagnostics. The overseer implements a monitoring server responsible for persisting data generated during the experiments in a database that can be later accessed by Nagios.

Semi-automated diagnosis: Causal Robotics DEbugging (CARE) [100]  is a two-phase method for diagnosing configuration faults (aka misconfigurations) in ROS-based systems. The method proposes first a causal model learning step, then an inference over the learnt model step. The causal model is three-layered and defines causal relations between (i) software-level configurations or hardware-level options (e.g., sensors), (ii) a map from configuration options to how they influence the outcome performance, and (iii) performance goals. The authors employ Fast Causal Inference (FCI) to extract the causal model from data. Moreover, the authors present an algorithm for causal path discovery (by inference) which finds a locally optimal path between configuration (aka symptom) and performance goal. Following a similar idea of three layered causal models, Kirchner D. et al. implemented RoSHA [101] an architecture for self-healing robotic systems implemented in ROS. Their approach relies on monitoring, diagnosis, recovery management and execution. In order to implement the automatic diagnosis, RoSHA relies on the standard tool `ros/diagnostics` to collect runtime failure information (i.e., w.r.t. component dependencies, communication, and time relations between components) to identify why the robot fails. The collected

information is regarded as symptoms, which are fed into a Bayesian network in order to compute the correlations between symptoms and root causes. Both techniques for diagnosis require structural constraints in the model to be feasible, asking for manual assistance to the diagnosis.

Multi-robot diagnosis: Morais et al. [102], promotes an approach for collaborative fault diagnosis using behavior trees in an agent programming language style, namely Rason  [102]. The approach relies on deploying an extra robot to assist a faulty robot in diagnosing a fault. Whenever the faulty robot identifies a verdict of fault in their behavior, the extra robot is activated to examine and provide extra information to disambiguate the diagnostic. For example, when a faulty robot does not identify that there is a mechanical issue with their wheel and *cannot reach their destination (verdict)*, the extra robot examines their wheel with a camera and identifies an issue. Following the analogy, the testing team may consider the deployment of extra robots to collect more information to assist with the diagnostics. Swarmlab [103] is an approach for debugging configuration bugs in swarm robotics. It abstracts the impacts of environment configurations (e.g., obstacles) on the drones in a swarm and automatically generates, validates, and ranks fixes for configuration bugs.

2.10.6 Strengths

A benefit of postmortem analyses in comparison to on-the-fly analyses is that the former, with full traces, presents better potential to improve the explanation precision [8].

2.10.7 Weaknesses

Diagnosis may require manual effort and domain knowledge to be effective, diagnosing without domain experience may lead to false positives.

2.11 AR2. Use reliable tooling in order to manage field data

2.11.1 Context (WHEN)

Maintaining the field resources is a fundamental step for the repeatability and reliability of field tests [4]. The field data is useful for generating field test cases and postmortem analyses. Yet, the data generated before, during and after testing must be kept and managed. Field data management often includes storing new data, annotating data, and generating reports. Such activities may turn intractable as the number of trials grows along with the complexity of the stored data (i.e., in perception systems, or learning-enabled systems). Manually managing such data is error-prone and sometimes unfeasible.

2.11.2 Reason (WHY)

Field testing is expensive and relying on ad hoc solutions to storing data may result in corrupted or incomplete field data. Unreliable tooling may be a source of flaky tests or skewed conclusions about the system (i.e., false-positive results).

10. <http://wiki.ros.org/diagnostics>

11. An official tutorial on analysis using the diagnostics package.

12. <https://www.nagios.org/>

2.11.3 Suggestion (WHAT)




The QA team should use reliable tools for field data management to avoid problems with corrupted, unreliable, and/or incomplete data. For example, the `warehouse_ros` package offers both MongoDB (git: `ros-planning/warehouse_ros_mongo`) and SQLite (git: `ros-planning/warehouse_ros_sqlite`) database backend for recording states, scenes, and messages. In addition, the Field Test Tool (git: `fkie/field_test_tool`) uses the PostgreSQL database manager extended with PostGIS for geolocalization.

2.11.4 Process (HOW)

Testers may use local workspace standard tooling aided by git-based versioning [87] or structured database [99], [104], and file-server [22], for field data management. On the one hand, using local workspace tooling is intuitive since tutorials in ROS traditionally rely on them, and most developers are used to the workspace management tools. On the other hand, structured approaches using database management tools may render easier querying and means to automatic report generation. When it comes to database integration for persisting long-term data, Malavolta et. al suggest using a dedicated node to interface the application and the database system in order to mitigate performance issues [10].

2.11.5 Exemplars

Local workspace with git support. Ortega A. et al. reports about the maintenance of test documents and artifacts for testing in the field [87]. The documents contain the specification of test procedures, reports, rosbag files, incident reports and cases in which people are detected. Test reporting is done on an in-house tool called Technology Readiness Level Test Report Library (TRL) and the reports are stored in a git repository where test incidents issues. TRL is based on the standard tooling for local workspace management in ROS, namely `wstool`¹³.

Database and File-server. The `warehouse_ros` package provides MongoDB (WRM ) and SQLite (WRS ) database backend for persistently recording states, scenes and messages, working in conjunction with RViz and the MotionPlanning plugin. The Field Test Tool (FTT ) provides support for logging, annotating, and automatically generating reports of field trials for autonomous ground vehicles [104]. The tool relies on a database to store field data, which is later accessed for automatic report generation. The authors use PostgreSQL database manager extended with PostGIS for geolocalization. The data is stored in the database through an interface ROS node which collects field data (Operation mode, GNSS positioning, local positioning, a map of the environment, and possibly images from a camera) from selected topics. Data collection is started and stopped in a web server that interfaces with the tester. The web interface can also be used to add annotations to collected field data. In addition to FTT, the Overseer architecture [99] enables the persistence of field data in a MySQL database, which can be further accessed by the monitoring system Nagios. On another stance, Hartsell et al. [22] studies ROS-based

systems with learned-enabled components. Often, testing such systems is prone to large dataset maintenance, which can be a problem in the field. The author's approach uses a mix of file servers for storing the data and a database for storing metadata. The file server relies on SSH File Transfer Protocol (SFTP) to store all generated data, resulting from the execution of a job (aka trial), results and notes from the experiments. The metadata is persisted in a version-controlled database for quick retrieval. As far as we are concerned, even though Hartsell's approach should work for field testing it was only validated in simulation testing [22].

2.11.6 Strengths

Tool support for persisting field data helps with data reliability since it is less likely that data will be lost or will be corrupted. In addition, this guideline enables traceability of trial execution with possible failure or fault.

2.11.7 Weaknesses

Using database systems to log long-term data may pose a threat to the performance of the testing apparatus.

REFERENCES

- [1] W. Schütz, *The testability of distributed real-time systems*. Springer Science & Business Media, 2007, vol. 245.
- [2] M. Beul, N. Krombach et al., "Autonomous navigation in a warehouse with a cognitive micro aerial vehicle," in *Robot Operating System (ROS)*. Springer, 2017, pp. 487–524.
- [3] M. Albonico, M. Djević et al., "Software engineering research on the robot operating system: A systematic mapping study," *Journal of Systems and Software*, vol. 197, p. 111574, 2023.
- [4] A. Bertolino, P. Braione et al., "A survey of field-based testing techniques," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–39, 2021.
- [5] K. Chaaban, "A new algorithm for real-time scheduling and resource mapping for robot operating systems (ros)," *Applied Sciences*, vol. 13, no. 3, p. 1532, 2023.
- [6] T. Lewis and K. Bhaganagar, "Configurable simulation strategies for testing pollutant plume source localization algorithms using autonomous multisensor mobile robots," *International Journal of Advanced Robotic Systems*, vol. 19, no. 2, p. 17298806221081325, 2022.
- [7] Z. Li, A. Hasegawa et al., "Autoware_perf: A tracing and performance analysis framework for ros 2 applications," *Journal of Systems Architecture*, vol. 123, p. 102341, 2022.
- [8] Y. Falcone, S. Krstić et al., "A taxonomy for classifying runtime verification tools," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 2, pp. 255–284, 2021.
- [9] M. Luckcuck, M. Farrell et al., "Formal specification and verification of autonomous robotic systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.
- [10] I. Malavolta, G. A. Lewis et al., "Mining guidelines for architecting robotics software," *Journal of Systems and Software*, vol. 178, p. 110969, 2021.
- [11] S.-Y. Jeong, I.-J. Choi et al., "A study on ros vulnerabilities and countermeasure," in *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, 2017, pp. 147–148.
- [12] V. Mayoral-Vilches, R. White et al., "Sros2: Usable cyber security tools for ros 2," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 11 253–11 259.
- [13] C. Hu, W. Dong et al., "Runtime verification on hierarchical properties of ros-based robot swarms," *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 674–689, 2019.
- [14] J. Huang, C. Erdogan et al., "Rosrv: Runtime verification for robots," in *International Conference on Runtime Verification*. Springer, 2014, pp. 247–254.

13. <http://wiki.ros.org/wstool>

- [15] A. Avizienis, J.-C. Laprie *et al.*, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.
- [16] A. Afzal, C. Le Goues *et al.*, "A study on challenges of testing robotic systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 96–107.
- [17] D. Bozhinoski, D. Di Ruscio *et al.*, "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective," *Journal of Systems and Software*, vol. 151, pp. 150–179, 2019.
- [18] S. Adam, M. Larsen *et al.*, "Towards rule-based dynamic safety monitoring for mobile robots," in *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings 4*. Springer, 2014, pp. 207–218.
- [19] M. Stadler and M. Vierhauser, "Romos: Flexible runtime monitoring support for ros-based applications," in *RoSE'23: 5th International Workshop on Robotics Software Engineering Proceedings*, 2023, p. xx.
- [20] S. Limsoonthrakul, M. N. Dailey *et al.*, "A modular system architecture for autonomous robots based on blackboard and publish-subscribe mechanisms," in *2008 IEEE International conference on robotics and biomimetics*. IEEE, 2009, pp. 633–638.
- [21] M. Efatmaneshnik and M. Ryan, "A study of the relationship between system testability and modularity," *INSIGHT*, vol. 20, no. 1, pp. 20–24, 2017.
- [22] C. Hartsell, N. Mahadevan *et al.*, "Model-based design for cps with learning-enabled components," in *Proceedings of the Workshop on Design Automation for CPS and IoT*, 2019, pp. 1–9.
- [23] E. B. Gil, R. Caldas *et al.*, "Body sensor network: A self-adaptive system exemplar in the healthcare domain," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2021, pp. 224–230.
- [24] F. Rovida, M. Crosby *et al.*, "Skiros—a skill-based robot control platform on top of ros," *Robot Operating System (ROS) The Complete Reference (Volume 2)*, pp. 121–160, 2017.
- [25] G. Rodrigues, R. Caldas *et al.*, "An architecture for mission coordination of heterogeneous robots," *Journal of Systems and Software*, vol. 191, p. 111363, 2022.
- [26] A. Albore, D. Doose *et al.*, "Skill-based design of dependable robotic architectures," *Robotics and Autonomous Systems*, vol. 160, p. 104318, 2023.
- [27] C. Hutchison, M. Zizyte *et al.*, "Robustness testing of autonomy software," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 276–285.
- [28] H. Choi, Y. Xiang *et al.*, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.
- [29] T. Blaß, D. Casini *et al.*, "A ros 2 response-time analysis exploiting starvation freedom and execution-time variance," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 41–53.
- [30] R. Halder, J. Proença *et al.*, "Formal verification of ros-based robotic applications using timed-automata," in *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormalISE)*. IEEE, 2017, pp. 44–50.
- [31] J. Bengtsson, K. Larsen *et al.*, "Uppaal—a tool suite for automatic verification of real-time systems," in *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*. Berlin, Heidelberg: Springer-Verlag, 1996, p. 232–243.
- [32] G. Conte, D. Scaradozzi *et al.*, "Development and experimental tests of a ros multi-agent structure for autonomous surface vehicles," *Journal of Intelligent & Robotic Systems*, vol. 92, no. 3, pp. 705–718, 2018.
- [33] K. Belsare, A. C. Rodriguez *et al.*, "Micro-ros," in *Robot Operating System (ROS) The Complete Reference (Volume 7)*. Springer, 2023, pp. 3–55.
- [34] A. Nordmann, R. Lange *et al.*, "System modes-digestible system (re-) configuration for robotics," in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. IEEE, 2021, pp. 19–24.
- [35] J. Staschulat, I. Lütkebohle *et al.*, "The rlc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*. IEEE, 2020, pp. 18–19.
- [36] P. Kaveti and H. Singh, "Ros rescue: fault tolerance system for robot operating system," *Robot Operating System (ROS) The Complete Reference (Volume 5)*, pp. 381–397, 2021.
- [37] A. Ferrando, R. C. Cardoso *et al.*, "Rosmonitoring: a runtime verification framework for ros," in *Annual Conference Towards Autonomous Robotic Systems*. Springer, 2020, pp. 387–399.
- [38] S. Elbaum and J. C. Munson, "Software black box: an alternative mechanism for failure analysis," in *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. IEEE, 2000, pp. 365–376.
- [39] A. Mitrevski, S. Thoduka *et al.*, "Deploying robots in everyday environments: Towards dependable and practical robotic systems," *arXiv preprint arXiv:2206.12719*, 2022.
- [40] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *Ieee transactions on software engineering*, vol. 32, no. 11, pp. 849–867, 2006.
- [41] U. Yayan and C. Baglum, "Tailored mutation-based software fault injection tool (im-fit)," *SoftwareX*, p. 101463, 2023.
- [42] R. Barbosa, J. Karlsson *et al.*, "Fault injection," *Resilience Assessment and Evaluation of Computing Systems*, pp. 263–281, 2012.
- [43] R. K. Lenka, S. Padhi *et al.*, "Fault injection techniques—a brief review," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE, 2018, pp. 832–837.
- [44] C. Bédard, P.-Y. Lajoie *et al.*, "Message flow analysis with complex causal links for distributed ros 2 systems," *Robotics and Autonomous Systems*, vol. 161, p. 104361, 2023.
- [45] M. Lahami, M. Krichen *et al.*, "Safe and efficient runtime testing framework applied in dynamic and distributed systems," *Science of Computer Programming*, vol. 122, pp. 1–28, 2016.
- [46] S. Silva, A. Bertolino *et al.*, "Self-adaptive testing in the field: are we there yet?" in *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2022, pp. 58–69.
- [47] S. Rivera and R. State, "Securing robots: An integrated approach for security challenges and monitoring for the robotic operating system (ros)," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 754–759.
- [48] B. Breiling, B. Dieber *et al.*, "Secure communication for the robot operating system," in *2017 annual IEEE international systems conference (SysCon)*. IEEE, 2017, pp. 1–6.
- [49] C. Schlegel, T. Haßler *et al.*, "Robotic software systems: From code-driven to model-driven designs," in *2009 International Conference on Advanced Robotics*. IEEE, 2009, pp. 1–8.
- [50] J. C. Kirchhof, J. Michael *et al.*, "Model-driven digital twin construction: synthesizing the integration of cyber-physical systems with their information systems," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 90–101.
- [51] E. Guerrero, F. Bonin-Font *et al.*, "Adaptive visual information gathering for autonomous exploration of underwater environments," *IEEE Access*, vol. 9, pp. 136 487–136 506, 2021.
- [52] T. Sotiropoulos, H. Waeselynck *et al.*, "Can robot navigation bugs be found in simulation? an exploratory study," in *2017 IEEE International conference on software quality, reliability and security (QRS)*. IEEE, 2017, pp. 150–159.
- [53] C. Saavedra Sueldo, I. Perez Colo *et al.*, "Ros-based architecture for fast digital twin development of smart manufacturing robotized systems," *Annals of Operations Research*, pp. 1–25, 2022.
- [54] A. Fend and D. Bork, "Cpsaml: a language and code generation framework for digital twin based monitoring of mobile cyber-physical systems," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2022, pp. 649–658.
- [55] J. Ernits, E. Halling *et al.*, "Model-based integration testing of ros packages: A mobile robot case study," in *2015 European Conference on Mobile Robots (ECMR)*. IEEE, 2015, pp. 1–7.
- [56] K. G. Larsen, M. Mikucionis *et al.*, "Testing real-time embedded software using uppaal-tron: an industrial case study," in *Proceedings of the 5th ACM international conference on Embedded software*, 2005, pp. 299–306.
- [57] N. Hammoudeh Garcia, M. Lüdtkke *et al.*, "Bootstrapping mde development from ros manual code - part 1: Metamodeling," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 2019, pp. 329–336.

- [58] N. Hammoudeh García, H. Deshpande *et al.*, “Bootstrapping mde development from ros manual code: Part 2—model generation and leveraging models at runtime,” *Software and Systems Modeling*, vol. 20, no. 6, pp. 2047–2070, 2021.
- [59] A. F. F. Santos, “Safety verification for ros software,” Ph.D. dissertation, Universidade do Minho, 2021.
- [60] M. Autili, L. Grunske *et al.*, “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar,” *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.
- [61] A. Santos, A. Cunha *et al.*, “The high-assurance ros framework,” in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. IEEE, 2021, pp. 37–40.
- [62] C. Lesire, S. Roussel *et al.*, “Synthesis of real-time observers from past-time linear temporal logic and timed specification,” in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 597–603.
- [63] D. Ničković and T. Yamaguchi, “Rtam: Online robustness monitors from stl,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 564–571.
- [64] S. Aldegheri, N. Bombieri *et al.*, “A containerized ros-compliant verification environment for robotic systems,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 222–225.
- [65] T. Yamaguchi, B. Hoxha *et al.*, “Rtam—runtime robustness monitors with application to cps and robotics,” *International Journal on Software Tools for Technology Transfer*, pp. 1–21, 2023.
- [66] A. Rodrigues, R. D. Caldas *et al.*, “A learning approach to enhance assurances for real-time self-adaptive systems,” in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2018, pp. 206–216.
- [67] C. Menghi, C. Tsigkanos *et al.*, “Specification patterns for robotic missions,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2208–2224, oct 2021.
- [68] —, “Mission specification patterns for mobile robots: Providing support for quantitative properties,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, p. 2741–2760, dec 2022.
- [69] S. Adam, M. Larsen *et al.*, “Rule-based dynamic safety monitoring for mobile robots,” *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 120–141, 2016.
- [70] R. Queiroz, D. Sharma *et al.*, “A driver-vehicle model for ads scenario-based testing,” *arXiv preprint arXiv:2205.02911*, 2022.
- [71] D. J. Fremont, E. Kim *et al.*, “Scenic: A language for scenario specification and data generation,” *Machine Learning*, pp. 1–45, 2022.
- [72] C. Xu, H. Huang *et al.*, “Sil simulation test platform based on prescan and ros,” in *2022 International Conference on Cyber-Physical Social Intelligence (ICCSI)*. IEEE, 2022, pp. 240–244.
- [73] A. Balakrishnan, J. Deshmukh *et al.*, “Percemon: Online monitoring for perception systems,” in *International Conference on Runtime Verification*. Springer, 2021, pp. 297–308.
- [74] X. Duan, Q. Wang *et al.*, “Implementing trajectory tracking control algorithm for autonomous vehicles,” in *2021 IEEE International Conference on Unmanned Systems (ICUS)*. IEEE, 2021, pp. 947–953.
- [75] S. B. Liu, H. Roehm *et al.*, “Provably safe motion of mobile robots in human environments,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1351–1357.
- [76] A. Dosovitskiy, G. Ros *et al.*, “Carla: An open urban driving simulator,” in *Conference on robot learning*. PMLR, 2017, pp. 1–16.
- [77] D. Tommaso, D. J. Fremont *et al.*, “Verifai: A toolkit for the design and analysis of artificial intelligence-based systems,” in *Proceedings of the 31st Conference on Computer Aided Verification, CAV 2019*, 2019.
- [78] M. Tideman, “Scenario-based simulation environment for assistance systems,” *ATZautotechnology*, vol. 10, no. 1, pp. 28–32, 2010.
- [79] S. Documentation, “Simulation and model-based design,” 2020. [Online]. Available: <https://www.mathworks.com/products/simulink.html>
- [80] E. Rohmer, S. P. N. Singh *et al.*, “V-rep: A versatile and scalable robot simulation framework,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 1321–1326.
- [81] S. Kim and T. Kim, “Robofuzz: Fuzzing robotic systems over robot operating system (ros) for finding correctness bugs,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 447–458.
- [82] Y.-S. Hsiao, Z. Wan *et al.*, “Mavfi: An end-to-end fault analysis framework with anomaly detection and recovery for micro aerial vehicles,” *arXiv preprint arXiv:2105.12882*, 2021.
- [83] B. Dieber, R. White *et al.*, “Penetration testing ros,” *Robot Operating System (ROS)*, p. 183, 2020.
- [84] E. T. Barr, M. Harman *et al.*, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.
- [85] A. Afzal, C. Le Goues *et al.*, “Mithra: Anomaly detection as an oracle for cyberphysical systems,” *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4535–4552, 2021.
- [86] A. Santos, A. Cunha *et al.*, “Property-based testing for the robot operating system,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 56–62.
- [87] A. Ortega, N. Hochgeschwender *et al.*, “Testing service robots in the field: An experience report,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 165–172.
- [88] C. Hildebrandt, S. Elbaum *et al.*, “Feasible and stressful trajectory generation for mobile robots,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 349–362.
- [89] S. Reid, “Software and systems engineering software testing part 1: Concepts and definitions,” ISO/IEC/IEEE 29119-1, Tech. Rep., 2013.
- [90] R. D. Caldas, A. Rodrigues *et al.*, “A hybrid approach combining control theory and ai for engineering self-adaptive systems,” in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020, pp. 9–19.
- [91] B. Zahn, J. Fountain *et al.*, “Optimization of robot movements using genetic algorithms and simulation,” in *RoboCup 2019: Robot World Cup XXIII 23*. Springer, 2019, pp. 466–475.
- [92] Z. Han, T. Williams *et al.*, “Mixed-reality robot behavior replay: A system implementation,” *arXiv preprint arXiv:2210.00075*, 2022.
- [93] Y. Koo and S. Kim, “Distributed logging system for ros-based systems,” in *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 2019, pp. 1–3.
- [94] C. Hildebrandt and S. Elbaum, “World-in-the-loop simulation for autonomous systems validation,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 10912–10919.
- [95] V. Estivill-Castro, R. Hexel *et al.*, “Continuous integration for testing full robotic behaviours in a gui-stripped simulation.” in *MoDELS (Workshops)*, 2018, pp. 453–464.
- [96] C. Pinciroli, V. Trianni *et al.*, “ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems,” *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.
- [97] L. Pitonakova, M. Giuliani *et al.*, “Feature and performance comparison of the v-rep, gazebo and argos robot simulators,” in *Towards Autonomous Robotic Systems: 19th Annual Conference, TAROS 2018, Bristol, UK July 25-27, 2018, Proceedings 19*. Springer, 2018, pp. 357–368.
- [98] C. Berger, “An open continuous deployment infrastructure for a self-driving vehicle ecosystem,” in *Open Source Systems: Integrating Communities: 12th IFIP WG 2.13 International Conference, OSS 2016, Gothenburg, Sweden, May 30-June 2, 2016, Proceedings 12*. Springer, 2016, pp. 177–183.
- [99] F. Roman, R. G. Maidana *et al.*, “Overseer: A multi robot monitoring infrastructure,” in *International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, 2018, Brasil., 2018.
- [100] M. A. Hossen, S. Kharade *et al.*, “Care: Finding root causes of configuration issues in highly-configurable robots,” *IEEE Robotics and Automation Letters*, 2023.
- [101] D. Kirchner, S. Niemczyk *et al.*, “RoshA: A multi-robot self-healing architecture,” in *Robot Soccer World Cup*. Springer, 2013, pp. 304–315.
- [102] M. G. Morais, F. R. Meneguzzi *et al.*, “Distributed fault diagnosis for multiple mobile robots using an agent programming language,” in *2015 International Conference on Advanced Robotics (ICAR)*. IEEE, 2015, pp. 395–400.
- [103] C. Jung, A. Ahad *et al.*, “Swarmbug: debugging configuration bugs in swarm robotics,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 868–880.

- [104] C. Tampier, A. Tiderko *et al.*, "Field test tool: automatic reporting and reliability evaluation for autonomous ground vehicles," in *2022 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE, 2022, pp. 73–78.