Thesis Proposal

Specification-Driven Detection of Misconfigurations in ROS-based Robot Software

Paulo Alexandre Canelas dos Santos

September 2025

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Robot Operating System, Misconfigurations, Domain Specific Language, Usability

Abstract

The Robot Operating System (ROS) is the most widely adopted open-source framework for developing robot software from reusable, off-the-shelf components. While its component-based architecture allows quick development of these systems, ensuring that components are correctly configured and integrated remains a major challenge. Documentation is often missing, outdated, or, when it exists, is not enforced, requiring developers to rely on implicit assumptions when configuring and integrating components. Misconfigurations arise from mismatched expectations and guarantees when configuring and integrating different components, leading these systems to unpredictable and dangerous behaviors. In this work, I take a step toward systematically understanding and addressing ROS misconfigurations. My thesis is that by understanding misconfigurations in ROS-based robot software, we can develop a usable domain-specific language for specifying component configurations and their integration to help detect real-world misconfigurations prior to deployment. To explore this thesis, I follow a four-step approach. First, I conduct a large-scale empirical study of developer-reported misconfigurations, from which I derive a taxonomy and dataset of real-world misconfigurations. Building on these insights, I then design and evaluate a domain-specific language, ROSpec, for specifying ROS components and systems. ROSpec allows developers to declare their user intent and context on components, allowing the detection of real-world misconfigurations prior to deployment. Next, to understand how these specifications hold over time, I analyze the evolution of ROS architectures in real-world projects and introduce a lightweight tool for detecting configuration and documentation drift. Finally, since ROS developers come from diverse technical backgrounds, I study the usability challenges of writing and maintaining such specifications in practice. In addition to these studies, this work contributes with publicly available datasets, specifications, and tools to support future research. Together, these contributions provide a foundation for detecting misconfigurations in component-based robotic systems, such as ROS, and take a step toward improving the reliability and maintainability of robot software.

Acknowledgments

This work was supported by Fundação para a Ciência e Tecnologia (FCT) and FEDER in the LASIGE Research Unit under the ref. (UID/00408/2025), RAP(EXPL/CCI-COM/1306/2021), RAP+ (LISBOA2030-FEDER-00663700), HPC (2025.00002.HPCVLAB.ISTUL, POR011PRE), the CMU Portugal Dual Degree PhD program (SFRH/BD/151469/2021), and a CMU Cylab seed grant.

Any opinions, findings, or recommendations expressed are those of the author(s) and do not necessarily reflect those of the US Government.

Contents

1 Introduction **1**
1.1 Thesis Statement . . . 3
1.2 Contributions . . . 3

2 Background **7**
2.1 Architecture . . . 7
2.2 Ecosystem . . . 8
2.3 Configuration . . . 9

3 Understanding Misconfigurations in ROS **11**
3.1 Study of Misconfigurations . . . 12
3.1.1 Methodology . . . 12
3.1.2 Threats to Validity . . . 15
3.1.3 Results . . . 15
3.2 Study of Existing Tools . . . 23
3.2.1 Methodology . . . 24
3.2.2 Threats to Validity . . . 25
3.2.3 Results . . . 25
3.3 Related Work . . . 27
3.4 Discussion . . . 28
3.5 Conclusion . . . 30

4 ROSpec: A Domain-Specific Language for ROS-based Robot Software **31**
4.1 Background and Motivating Example . . . 32
4.2 Related Work . . . 34
4.3 Language Properties . . . 35
4.3.1 Methodology . . . 35
4.3.2 Threats to Validity . . . 38
4.4 ROSpec Design . . . 39
4.4.1 Node definitions, Parameter Refinements and Dependencies . . . 39
4.4.2 Messages Alias and Field Definition . . . 41
4.4.3 Publisher & Subscriber, Service and Action Connections . . . 42
4.4.4 Quality of Service (QoS) and Color Format Policies . . . 43
4.4.5 TF Transforms and Contextual Information . . . 44

4.4.6 Plugins 46
4.5 Language Semantics 47
4.5.1 Grammar 47
4.5.2 Formation Rules 48
4.6 Evaluation 50
4.6.1 Case Study: Neobotix MP-400 in AWS Small Warehouse 50
4.6.2 Misconfiguration Modeling 53
4.7 Discussion & Future Work 56
4.8 Conclusion 59

5 Architectural Evolution and Drift Analysis in Open Source Robotics (Proposed Work) 61
5.1 Methodology 62
5.1.1 Data Collection and Analysis 64
5.1.2 Threats to Validity 65
5.2 Proposed Approach 65
5.2.1 Architectural Recovery from Source Code 66
5.2.2 Documentation Architectural Inference 68
5.2.3 Drift Analysis 69
5.3 Conclusion 70

6 Specifying ROS-bots with ROSpec: A Usability Study (Proposed Work) 71
6.1 Study Design 72
6.1.1 Methodology 73
6.1.2 Threats to Validity 74
6.2 Conclusion 75

7 Proposed Timeline 77
7.1 Timeline 77

8 Conclusion 79

Bibliography 81

Complete Language Semantics 95
.1 Grammar 96
.2 Formation Rules 97

List of Figures

2.1 Structural architectural partial view of a publisher subscriber model in the Fetch demo system with three nodes and four topics. 7
2.2 Structural architectural partial view of services of the Fetch demo robotic with one service, two nodes and two topics. 8

3.1 Three-step methodology for analyzing ROS Answers questions. 13
3.2 Example of a ROS Answers question. 14
3.3 Mindmap of the misconfigurations identified from the study. 16
3.4 Example of a Name Mismatch from ROS Answers. 17
3.5 Example of a ROS Answers question behavioral misconfiguration. 20
3.6 Methodology of the literature review. 24

4.3 Faulty branch of TF tree due to missing broadcast from `base_link` to `laser`. 45
4.7 AWS Small Warehouse World. 50
4.8 Neobotix MP-400. 50

5.1 Architectural view of the publisher-subscriber graph documented in the mgonzs13/yolo_ros project. The package was refactored and references to *v8* were removed. However, current documentation does not reflect the implementation. 62
5.2 Overview of our approach for detecting architectural drift in ROS systems. 66

6.1 Overview of the usability study methodology. 74

1 Type Formation Rules. 97
2 Context Formation Rules. 97
3 Expression Formation Rules. 98
4 Statement Formation Rules. 99
5 Definition Formation Rules. 100
6 Subtyping Rules. 101
7 Auxiliary rule to check QoS settings. 101

List of Tables

3.1 Overview of each technique of the literature review. 26

4.1 Overview of the main domain-specific concepts in ROS 2. 36
4.2 Overview of configuration and integration properties. 37
4.3 Summary of results from manual analysis and specification of 182 questions 54

Listings

2.1 Launch file of a joystick control for the Fetch robotic system in ROS 1. . . 9
3.1 Example where `tf_prefix` parameter required for operating with multiple robots is **`Missing`**, leading the system to use the **`Default Parameter`** value. . 18
4.1 Writer specification example with parameter refinements. 40
4.2 Integrator specification with a dependency mismatch. 40
4.3 Writer specification of message and type aliases. 41
4.4 Specification for nodes and their connections. 42
4.5 Integrator specification with a missing publisher misconfiguration. 42
4.6 OpenNI and custom node specification with QoS policies 44
4.7 System configuration with QoS and color format misconfigurations 44
4.10 Node and plugin specification definition example 46
5.1 Excerpt of source-code from mgonzs13/yolo_ros for yolo_node. The highlighted ROS API calls create a component, its configuration and connections. 67
5.2 Tree-sitter query to capture a lifecycle node initialization in Python for ROS 2. 68

Chapter 1

Introduction

Robot software development is a rapidly growing field [12] with applications in food delivery [103], agriculture [7], and surgery [110]. Despite its momentum, building and testing these systems is time-consuming and requires expertise and coordination across various domains, including software and mechanical engineering, mathematics, and physics [1].

The Robot Operating System (ROS) [94, 112] manages this complexity through a component-based architecture that promotes reuse. ROS is an open-source framework that allows developers to focus on integrating and configuring reusable, off-the-shelf components [42, 78]. Since its release, ROS has become the *de facto* most popular open-source framework for developing robot software, with its consortium including major companies such as Bosch, Caterpillar, and Microsoft [118].

Components are processes that receive, process, and may produce information to other components. These are configured through source code and multiple configuration files, often scattered across the system. Due to ROS's loosely coupled architecture, these components can be easily configured and integrated into complex robotic systems. For example, ROS provides components for autonomous navigation [157], obstacle detection [9], and translating motion sequences into motor commands [27, 30].

Despite the relative ease of integrating components using configuration files in ROS, *correctly* configuring systems presents a considerable challenge. Documentation is often missing or outdated, making it difficult to understand how components should be used and integrated [42, 53]. As a result, developers are forced to inspect components source code and configuration files directly. Moreover, since component configurations may depend on one another, developers must rely on the time-consuming task of experimentation [29] (e.g., through simulation or field testing [1]) to verify that the system is correctly configured. As systems grow in size and complexity, with more components and connections, the lack of documentation makes proper configuration even more challenging.

Therefore, developers often have to rely on implicit, unverified and outdated assumptions when configuring and integrating components, which may lead to misconfigurations. *Misconfigurations* in ROS result from mismatched expectations and guarantees when configuring and integrating different components. For instance, components may only be used when the system is executed in specific environments,[1] or expectations regarding data are different between components — a component may provide RGB images, while another expects grayscale images, leading to color format misconfigurations.[2]

Prior work has addressed specific types of misconfigurations through general assumptions about component usage. HAROS [125] and ROSDiscover [133] recover system architectures to detect missing or inconsistent connections between components (i.e., structural misconfigurations). ROSInfer [40] infers component behavior (reactive, periodic, and state-dependent) to identify behavioral misconfigurations. Phys [72] focuses on physical unit mismatches by using ROS assumptions. Yet, structural, behavioral, and physical unit mismatches represent only a subset of possible misconfiguration categories currently unknown in ROS. Moreover, these approaches lack an oracle that defines the correct configuration and integration of components to help them detect these misconfigurations.

The purpose of my work is to improve the quality assurance of highly configurable component-based robot software. As systems that interact with the physical world, ensuring correct component configuration and integration is critical to prevent unpredictable and dangerous robot behaviors. Since ROS is the most widely adopted robotics framework, I use it as a proxy for broader component-based robot software development. To this end, I study ROS's evolving ecosystem, identify sources of misconfiguration, and propose an oracle in the form of a domain-specific language to help in misconfiguration detection.

Existing work has primarily focused on errors in autonomous vehicles and robotic systems and provides little insight into "harder-to-detect" misconfigurations. For instance, in autonomous vehicles, nearly a third of reported bugs are configuration errors [52], and similar issues have been observed in Unmanned Autonomous Vehicles (UAV) [142]. Studies also report dependency problems [47], physical unit mismatches in ROS projects [104], and errors in URDF/Xacro files [2]. ROBUST [131, 132] catalogued a wider range of errors, including missing runtime dependencies and naming errors. However, no existing study focuses on studying the different categories of misconfigurations, more specific to highly configurable, component-based systems like ROS and how to address them effectively.

Domain-specific languages (DSLs) have been proposed as a way to detect errors, but existing approaches often lack the domain and usability requirements needed to effectively address misconfigurations. In robotics, DSLs have been used to specify and verify component interconnections [58, 101, 105, 144], real-time requirements [38, 49, 90, 96, 114], and hardware-software relationships [49, 126, 144]. However, the sucessful adoption of these tools depends on several factors: expressiveness to capture system architectures [69], domain specialization to simplify descriptions [147], intuitive syntax [97], and abstraction levels that match developer understanding [147]. To the best of my knowledge, no language has been designed to meet these requirements in the context of detecting ROS misconfigurations.

Prior work has studied documentation drift in other software domains where stale documentation leads to incorrect assumptions and bugs [68, 119, 128]. ROS systems face similar challenges where components evolve, documentation becomes outdated, causing developers to misconfigure systems based on incorrect information. However, ROS-based systems

[1] https://answers.ros.org/question/185909
[2] https://answers.ros.org/question/201031

presents an additional challenge beyond parameter-level documentation. ROS systems define architectures through component interactions, topic connections, and transformation trees that must also be documented and maintained. Yet no prior work has studied how ROS architectures evolve over time, how frequently documentation drifts from implementation, or which architectural elements are most prone to drift. Understanding these patterns is critical for understanding if specification-driven approaches can be sustained as systems grow in complexity.

To address the challenges of understanding and detecting misconfigurations, my work follows a four-step approach. First, I conduct a qualitative study to categorize the types of misconfigurations developers face and assess the extent to which current techniques address them. This analysis highlights gaps in existing work and guides future research on tools and techniques to detect misconfigurations. Second, I design a domain-specific language for specifying ROS components and systems, allowing developers to verify configurations prior to deployment. This DSL serves as an oracle that adapts to developer intent and context, verifying components and their integration without the time-consuming burden of simulation [29] or field testing [1]. Third, I evaluate the evolution of ROS architectures over time and introduce a tool to detect architectural and documentation drift. Understanding this evolution is critical for maintaining specifications and warning developers when documentation or oracles diverge from implementation. Finally, I study the usability challenges of writing and maintaining specifications in robotics. Since robot software developers come from diverse backgrounds [1], specification languages must be accessible, adaptable, and maintainable across different types of users.

Despite significant progress, the robotics community continues to face major challenges in verifying component-based robot software, such as ROS. Existing approaches focus on well-known issues, such as incorrect component connections, but little is understood about the broader spectrum of misconfigurations developers face or how to address them. This work takes the first step in the robotics software engineering domain towards building that understanding and developing techniques to address these misconfigurations.

1.1 Thesis Statement

By understanding misconfigurations in ROS-based robot software, we can develop a usable domain-specific language for specifying component configurations and their integration to help detect real-world misconfigurations prior to deployment.

1.2 Contributions

In this work, I propose a set of qualitative and quantitative studies to better understand the categories of misconfigurations developers face, the usability challenges encountered by ROS developers when writing formal specifications, and the ways in which ROS components and systems evolve over time. On the qualitative side, I perform a manual analysis of thousands of misconfigurations reported on a popular Q&A platform to create a taxonomy and dataset of real-world misconfigurations. I also propose to conduct interviews with ROS developers to explore the challenges they face in writing and maintaining specifications given their different backgrounds. On the quantitative side, I conduct a large-scale study of configuration and architectural evolution and drift in ROS-based systems using real-world projects mined from GitHub.

Furthermore, this work also introduces an approach for specifying ROS components and systems to detect misconfigurations, as well as an approach for automatically identifying configuration and architectural drift between documentation and implementation. To evaluate the domain-specific language I take a two step approach: (1) I specify a case study on a warehouse robotic system from a widely used ROS learning platform; and (2) specify components to detect real-world misconfigurations from a dataset of misconfigurations. To evaluate the drift analysis tool, I use real-world ROS projects mined from GitHub.

This thesis proposes to contribute in the following ways:

1. It identifies categories of misconfigurations that developers face when building ROS-based systems;
2. It examines the gap in current approaches for detecting these misconfigurations;
3. It presents a domain-specific language for detecting ROS misconfigurations;
4. It studies the evolution of ROS architectures to understand the potential maintenance effort of specifications;
5. It analyzes the co-evolution of architecture and documentation and provides warnings for configuration and architectural drift;
6. It identifies usability requirements for domain-specific languages based on the needs of roboticists from diverse backgrounds.

Additionally, this thesis proposes to contribute by publicly providing the set of tools and datasets:

- A codebook with the taxonomy of misconfigurations identified in the empirical study: https://doi.org/10.5281/zenodo.12643079;
- A dataset of Q&A questions annotated with misconfiguration categories: https://doi.org/10.5281/zenodo.12643079;
- A domain-specific language, publicly available and documented, for specifying components and systems and detecting misconfigurations: https://rospec.pcanelas.com;
- The specification of components in a case study, along with partial specifications for detecting real-world misconfigurations: https://doi.org/10.5281/zenodo.15722060;
- A lightweight analysis tool to automatically infer configuration and architectural evolution and drift in ROS-based robot software;
- A dataset of repositories with their inferred architectures, including an analysis of architecture evolution and drift.

Overall, this thesis identifies the types of misconfigurations in robotic systems and studies the evolution of configurations and architectures in ROS systems, while proposing a domain-specific language to address these misconfigurations in a usable way. Together, these studies and approaches take a step toward improving the quality of robotic systems.

Parts of this thesis have been published in peer reviewed venues:

- **Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches** [21]. Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. International Symposium on Software Testing and Analysis. 2024.
- **ROSpec: A Domain-Specific Language for ROS-based Robot Software** [24]. Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. Proceedings of the ACM on Programming Languages, OOPSLA. 2025.

Chapter 2

Background

Component-based robot software allows developers to quickly compose and deploy their system by integrating reusable components [6]. Indeed, in recent years several frameworks, such as CARMEN [99], OROCOS [15], YARP [98], and the Robot Operating System (ROS) [112] have been introduced to improve robotics development. Among these, ROS has become the de facto open-source framework for developing robot software, with its consortium including large industry companies such as Bosch, CAT, and Microsoft [118]. In this section, I provide a high-level introduction to ROS as a middleware for building component-based robotics software and as an open-source ecosystem of reusable components.

2.1 Architecture

Systems in ROS are built as a collection of independent processes, known as *nodes* or *components*, each responsible for providing certain functions (e.g., perception, planning, control, driver interfacing). At its core, ROS's responsibility is to provide the "plumbing" that facilitates communication between distributed components.

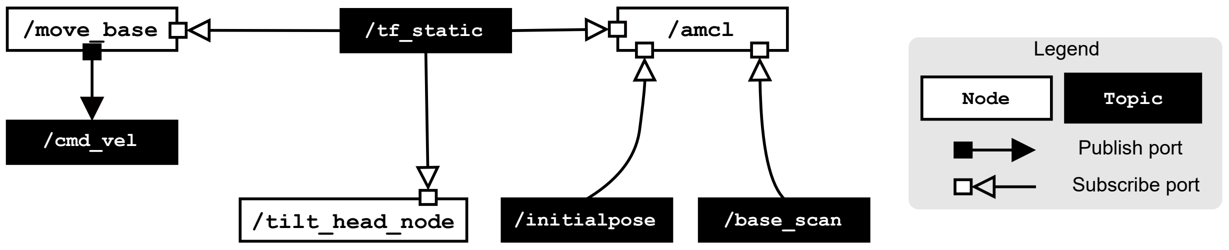


Figure 2.1: Structural architectural partial view of a publisher subscriber model in the Fetch demo system with three nodes and four topics.

The bulk of communication within ROS follows an anonymous publish/subscribe pattern [122], followed by synchronous and asynchronous remote procedure calls (i.e., services and actions, respectively). At one end, components (e.g., a camera driver) publish messages to named topics (e.g., `/camera/color/image_raw`). On the other hand, components (e.g., object detection) subscribe to those same topics to receive messages. Neither the publishers nor the subscribers are directly aware of one another's identities (i.e., they are spatially decoupled [43]). Moreover, communications are defined at run-time via calls to the ROS API (e.g., strings are used to state topics by name). For instance, Figure 2.1 depicts an example of a publish/subscribe system in ROS for the Fetch system. The `cmd_vel` and `initialpose` topics exchange information on a specific type of messages. In this case, `cmd_vel` receives messages of type `Twist`, while `initialpose` receives messages from type `PoseWithCovarianceStamped`. Each node publishes information to a topic queue at a specific rate, and consumes information from the topic.

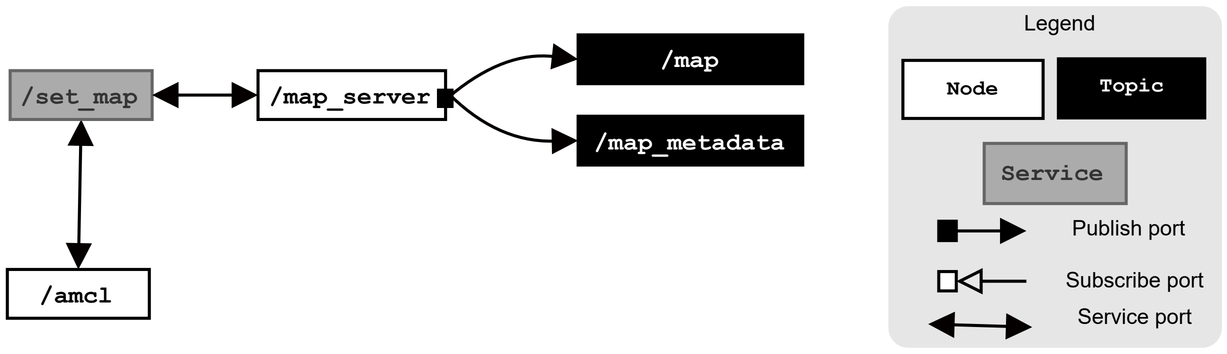


Figure 2.2: Structural architectural partial view of services of the Fetch demo robotic with one service, two nodes and two topics.

ROS also provides synchronous and asynchronous remote procedure calls, i.e., services and actions. Services allow the developers to have components request information to a server, and wait for the response deliver, while actions continues the execution and act upon received information. For example, Figure 2.2 presents a partial structural architectural view of a service for the Fetch demo system. In this case, the node `map_server` provides a service through the `set_map` topic, while `amcl` is a client through the same topic. While the server processes the message, the client halts the execution. At the end of the server computation, the server creates a message response and sends it back to the client, which immediately resumes its execution. The presented view abstracts the internal details of the communication, where the the `set_map` topic is split into a request and response topics. Similarly, actions in ROS communicate through three topics that dispatch the request, response and feedback while the server is executing.

2.2 Ecosystem

One of ROS's biggest strengths is its rich ecosystem of generic open-source components that can be reused for common robot functions [77, 92]. For example, MoveIt! [30] provides motion planning and execution for manipulation, `ros_control` [28] provides implementations of low-level controllers (e.g., velocity, joints, effort), and `ros_localization` [100] provides filters for state estimation (e.g., by fusing GPS and IMU data). By reusing off-the-shelf

```
 1 <launch>
 2     <arg name="dev" default="/dev/input/js0" />
 3     <!-- Launch joy node -->
 4     <node pkg="joy" type="joy_node" name="joy">
 5         <!--Customize this to match the location of your joystick-->
 6         <param name="dev" value="$(arg dev)" />
 7         <param name="deadzone" value="0.2" />
 8         <param name="autorepeat_rate" value="40" />
 9         <param name="coalesce_interval" value="0.025" />
10     </node>
11     <!-- Launch python interface -->
12     <node pkg="moveit_ros_visualization"
13         type="moveit_joy.py"
14         output="screen"
15         name="moveit_joy" />
16 </launch>
```

Listing 2.1: Launch file of a joystick control for the Fetch robotic system in ROS 1.

components, developers can, in theory, reduce the cost and complexity of building robot software. However, as these components are generic, they must be configured to work in specific contexts.

Recently, Open Robotics released ROS 2 [94], addressing limitations in ROS 1. While maintaining conceptual compatibility, ROS 2 introduces changes such as, DDS-based communication for improved reliability and real-time performance; Quality of Service (QoS) policies for fine-tuned message delivery control; improved security features and multi-platform support; and lifecycle management for deterministic node behavior.

Despite these improvements, both ROS versions share the fundamental challenge of correct configuration. The dynamic, loosely coupled architecture means developers must follow assumptions and conventions when integrating components, assumptions that still neither enforced nor documented [1, 42].

2.3 Configuration

ROS uses configuration files to customize and arrange those components into a functioning ensemble. These include Launch XML (or Python) files, which launch and compose each component within the system, and ROS Parameter YAML files, which are used by components at run-time to customize their behavior (e.g., specifying a color format and topic name for camera images). For example, Listing 2.1 presents the launch file code for controlling the Fetch robotic system with a joystick in ROS 1.[1] Launch files can define arguments taken as input when executing the command, such as the device path for the joystick/gamepad (line 7). Furthermore, they also define parameters, needed for the system execution (lines 8 to 10), and declares the node name responsible for moving the system.

[1] /ZebraDevs/fetch_ros/blob/melodic-devel/fetch_moveit_config/launch/joystick_control.launch

General-purpose components use these configuration files to tailor their behavior to a particular robot, application, or environment. Particularly complex and variable components and subsystems (e.g., Nav2, MoveIt!, ros_control) go beyond providing a fixed set of parameters and embed a domain-specific language within the ROS parameter system.

As this thesis provides insights into the ROS ecosystem, its evolution, and sources of misconfiguration, I provide additional background specific to each topic in the corresponding chapters.

Chapter 3

Understanding Misconfigurations in ROS

To provide plug-and-play functionality, ROS components are typically required to make assumptions about the context in which they are used (e.g., a topic should receive messages of the correct type at a certain frequency) that are neither checked nor documented [42]. Misconfigurations occur when one or more components make different, conflicting assumptions about the robot, leading to unintended and potentially dangerous behavior (e.g., property damage, human harm) during deployment. Given the importance of safety within this domain, it is vital to identify misconfigurations before the robot is deployed, and as early as possible. To that end, the robotics software engineering community has begun to develop tools to detect certain misconfigurations, such as those related to physical units [72], architecture [133], and reference frames [73].

To systematically tackle the misconfiguration problem, it is critical to understand the types of misconfigurations that occur in the wild and whether existing tools are designed to detect them. Based on our own experiences with ROS, we know that physical units, architectural, and reference frames are not ROS's only categories of misconfiguration. Software misconfigurations have been thoroughly studied in different contexts of software development (e.g., security [41, 113], databases [88, 154], cloud computing [67, 137, 156] and networks [95]). However, misconfigurations within ROS are inherently different due to their cyber-physical nature.

In this chapter, we set out to **identify the broader set of misconfigurations that impact ROS systems and to determine which detection techniques address them and which misconfiguration types are going undetected**. This knowledge can guide future research in the robotics software engineering community in developing novel tools and techniques to address them.

We first derive a taxonomy of misconfigurations within ROS systems by conducting an empirical study of relevant questions posted to ROS Answers, a ROS-specific Q&A site similar to Stack Overflow. Secondly, we determine the extent to which state-of-the-art analysis tools help to address those misconfigurations by conducting a literature review of analysis papers published at several major software engineering, architecture, testing, and robotics conferences. Finally, as part of our analysis, we highlight misconfigurations unaddressed by existing techniques and further discuss research opportunities in developing new techniques for them.

Through our study, we make the following contributions:

- A taxonomy of misconfigurations in ROS, based on a qualitative study of a popular Q&A platform (Section 3.1.3);
- A literature review of the state-of-the-art approaches and how they cover the misconfigurations (Section 3.2);
- A dataset of misconfigurations and questions manually analyzed and categorized that can be used to guide future studies and develop novel techniques [19].

3.1 Study of Misconfigurations

To understand which misconfigurations current tools can address, we first need to know the different types that exist. Therefore, we ask the following research question:

RQ1: What kinds of misconfigurations do developers make when building robot software systems with ROS?

To answer this question, we perform an empirical study of ROS Answers, the, until recently, primary Q&A forum for ROS [42].[1] Q&A platforms are designed for users to post their problems, including explaining the scenario where the problem occurs. While only some of the questions pertain to misconfiguration, we found many such instances in a prospective search. Q&A websites provide more detailed misconfiguration examples than social coding platforms (e.g., GitHub) since commit messages often lack important context, making it difficult or impossible to reliably identify misconfigurations. Furthermore, issue trackers often describe bugs in individual components rather than the difficulties of integrating those components into a working system. To that end, we perform a thematic analysis of questions posted to ROS Answers.

In the rest of this section, we describe our methodology (Section 3.1.1) and its associated threats to validity (Section 3.1.2) before presenting our taxonomy (Section 3.1.3).

3.1.1 Methodology

Figure 3.1 outlines our high-level methodology, which takes inspiration from studies of similar Q&A platforms [2, 130]. Below, we describe each step of our methodology.

Data Collection & Filtering. We first gathered all 67 189 questions posted to ROS Answers between January 1st, 2011, and November 20th, 2022. Figure 3.2 presents an example of a question (231458) and its accepted answer. We then filtered these questions to a set of 27 547 by selecting only those with an accepted answer as they offer an alternative perspective and accepted solution from the user.

During the second step, we narrowed the accepted questions to those referring to ROS and architectural concepts, expecting to obtain misconfiguration questions. Reducing the set of questions is a common practice in the literature [2, 3, 130]. We used common ROS concepts as defined in the ROS Wiki: node, subscribe topic, message, parameter, service, action, launch, publish, and subscribe. Since developers indirectly define their system's

[1]In August 2023, Open Robotics moved to the Robotics Stack Exchange (robotics.stackexchange.com).

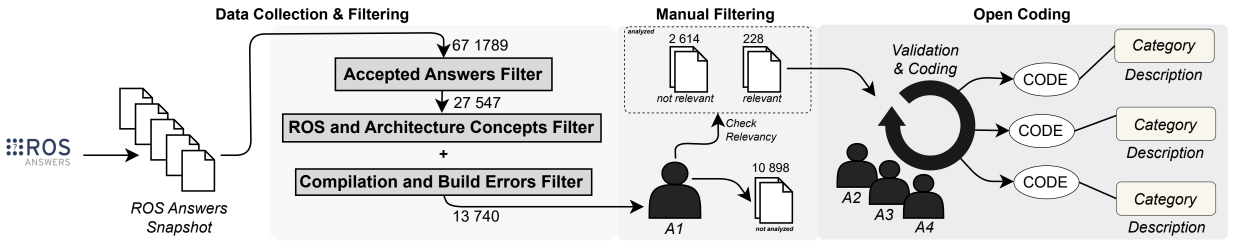
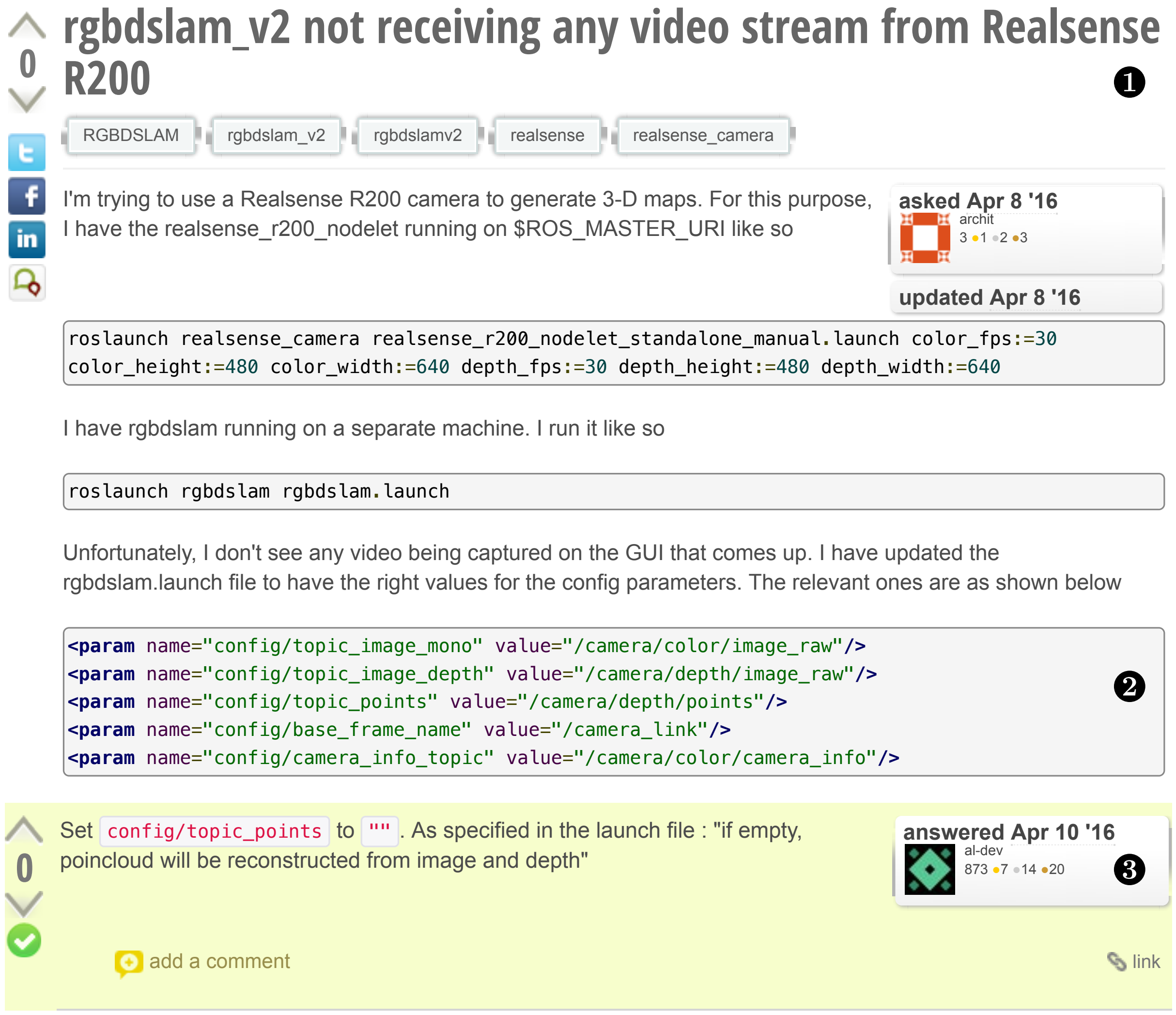


Figure 3.1: Three-step methodology for analyzing ROS Answers questions. **Step 1** collects a snapshot of 67 189 questions. **Step 2** selects questions with accepted answers containing ROS or software architecture concepts and filters questions about compilation or building issues. **Step 3** produces a sample of 228 questions filtered based on their relevance. Questions are divided into stages, where their codings are iteratively improved.

architecture when changing configurations, we also identify questions related to software architecture errors [55]: architecture, mismatch, assumption, incompatibility, inconsistency, integration, and configuration. Subsequently, the set was further refined by removing questions containing keywords related to installation and build errors (e.g., **`build error`** or **`compil*`**). Build errors are comparatively easy to determine and diagnose, as developers can examine the error messages generated. This work focuses on undetected misconfigurations during the deployment process, subsequently impacting robots behaviors. These errors are significantly more challenging to detect and trace back to their source. This final filtering step provided a total of 13 740 accepted questions considered for sampling.

Manual Filtering. The first author (A_1) randomly selected questions from the 13 740 questions and labeled them as relevant or irrelevant by analyzing their content to determine if it described a failed attempt to configure the system (i.e., interacting with the configurations or source code files).

Open Coding. The first author (A_1) iteratively provided sets of relevant questions to three other authors (A_2, A_3, A_4) who applied an open coding [44] approach to construct a taxonomy of misconfigurations. At each step, the authors individually proposed an updated set of codes for their sub-set of questions before discussing those codes and merging them into a revised taxonomy. The subsets were constructed such that each question was analyzed by three different authors, allowing a diversity of perspectives to be captured and reducing the author bias. This process continued until reaching the saturation point [57] (i.e., no further changes were made to the taxonomy after reaching the end of a step) after analyzing 228 questions, resulting in a final taxonomy of 12 categories and 50 subcategories of ROS misconfiguration.


rgbdslam_v2 not receiving any video stream from Realsense R200 ❶

RGBDSLAM rgbdslam_v2 rgbdslamv2 realsense realsense_camera

I'm trying to use a Realsense R200 camera to generate 3-D maps. For this purpose, I have the realsense_r200_nodelet running on $ROS_MASTER_URI like so

asked Apr 8 '16
archit
3 •1 •2 •3

updated Apr 8 '16

```
roslaunch realsense_camera realsense_r200_nodelet_standalone_manual.launch color_fps:=30
color_height:=480 color_width:=640 depth_fps:=30 depth_height:=480 depth_width:=640
```

I have rgbdslam running on a separate machine. I run it like so

```
roslaunch rgbdslam rgbdslam.launch
```

Unfortunately, I don't see any video being captured on the GUI that comes up. I have updated the rgbdslam.launch file to have the right values for the config parameters. The relevant ones are as shown below

```
<param name="config/topic_image_mono" value="/camera/color/image_raw"/>
<param name="config/topic_image_depth" value="/camera/depth/image_raw"/>
<param name="config/topic_points" value="/camera/depth/points"/>
<param name="config/base_frame_name" value="/camera_link"/>
<param name="config/camera_info_topic" value="/camera/color/camera_info"/>
```
❷

Set `config/topic_points` to `""`. As specified in the launch file : "if empty, poincloud will be reconstructed from image and depth"

answered Apr 10 '16
al-dev
873 •7 •14 •20 ❸

Figure 3.2: Example of a ROS Answers question corresponding to a misconfiguration where the developer incorrectly defined a parameter value. Each question contains a title (❶), content with text and source code (❷), and metadata about the author, date, and number of votes. Questions may include comments, answers, and an accepted answer (❸).

Labeling. Finally, we labeled each of the 228 questions using the final taxonomy. Half of the questions were labeled by one pair of authors (A_1, A_2) and the other half by a different pair (A_3, A_4). We calculated the agreement by dividing the number of codes both authors agreed on by the total number of codes used. This first step led to an agreement of 84.12% and 85.5% for each pair of authors. Then, each pair compared the codes that differed by one code and adjusted their code if in agreement. Furthermore, to determine documentation-related questions, the authors collected all questions annotated with documentation, discussed random instances of these, and re-annotated all questions until they reached an agreement on using this code. Finally, the authors discussed 26 questions with initial disagreements. The authors who did not analyze a given question were arbiters during the discussion.

3.1.2 Threats to Validity

External Validity. We identify two primary external validity threats: the generality of our results to (a) different ROS versions and distributions and (b) expert users. The first threat relates to the possible predominance of ROS 1 over ROS 2 questions and the impact of ROS distributions. Given the relatively recent release of ROS 2 [93] in 2018, we expected more questions related to ROS 1. We found that the analyzed questions rarely specified ROS or distribution versions (37 out of 228), making it infeasible to determine version-specific misconfigurations. Furthermore, as there are few architectural differences between ROS versions, we believe that our findings generalize to both versions. The second threat concerns the applicability of our findings to real-world scenarios. By sampling data from a popular ROS Q&A platform, we are addressing real-world developer issues. Nevertheless, we recognize that industrial settings may present unique, undisclosed misconfigurations. Our taxonomy provides a basis for further studies in such contexts.

Internal Validity. We identify four main threats to internal validity: the initial samping method, the generalizability of the sampled data, biases in question analysis, and potential misrepresentation of misconfiguration types in the sampled ROS Answers questions. The first threat regards the initial sampling and validation step with only one author, possibly introducing personal biases in the selection. To mitigate this step, we performed a preliminary study similar to the current one, in which all authors looked at a sample of questions and validated them as relevant or not relevant. To address the second threat, we sampled relevant ROS Answers questions, inspected, validated, and categorized them until reaching theoretical saturation, preventing the introduction of unaddressed categories with new questions. For the third threat, we iteratively analyzed questions and validated their relevance and misconfiguration categories with at least two authors. The forth threat concerns the sample's representativeness. We focus on questions related to ROS concepts and software architecture, which are more likely to contain misconfigurations. While other misconfigurations may be overlooked, this does not invalidate the identified categories.

3.1.3 Results

In this section, we describe each high-level category and sub-category of `Misconfiguration` in detail, along with relevant examples. Figure 3.3 depicts the mindmap of the taxonomy of misconfigurations.

Messages. Most ROS communication occurs via messages exchanged between components over named topics. Through our analysis, we identified five misconfigurations related to messaging.

`Format` misconfigurations arise due to a mismatch between two or more components in the message format that are exchanged on a shared topic. For instance, the `initialpose` topic, representing the initial position and orientation of the robot in the map, accepts `geometry_msgs/Pose` messages. Both publishers and subscribers must respect the message format when exchanging messages on this topic. However, a misconfiguration occurs when either end breaks the "contract" and expects a different type of message.

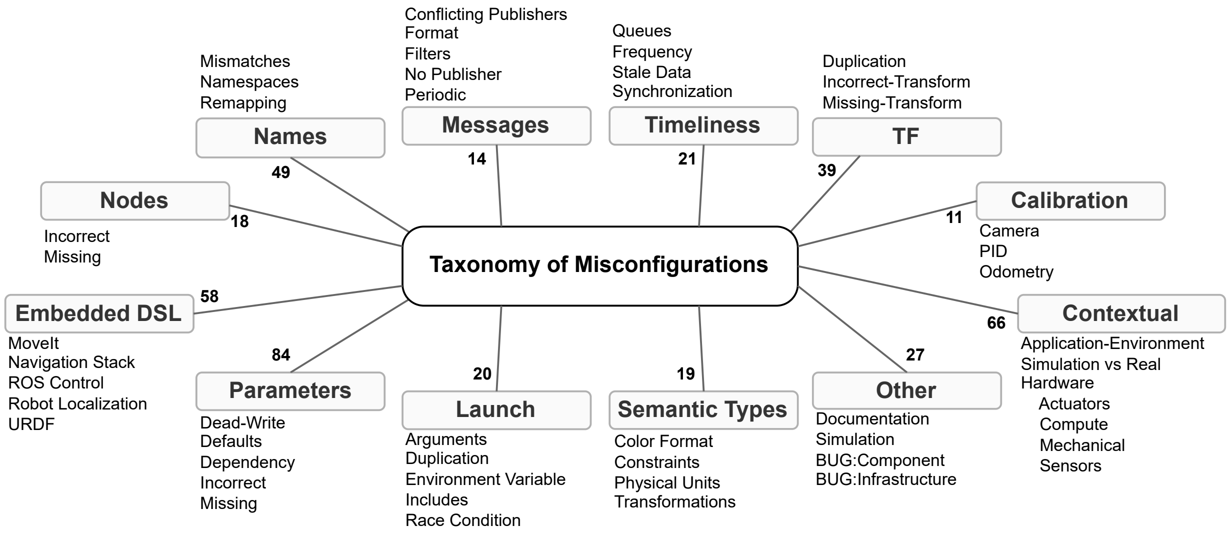


Figure 3.3: Mindmap of the misconfigurations identified from the study presenting the 12 high-level categories of misconfigurations, and their 50 sub-categories level. Each misconfiguration contains the number of questions annotated with the code. Each question may refer to more than one misconfiguration.

Components may expect a specific number of publishers to the topics to which they subscribe. This assumption eases components' expectations of the frequency of messages they receive. `No Publisher` misconfigurations occur when a component subscribes to a topic no publisher sends messages to. The subscriber waits indefinitely for messages that never arrive. Figure 3.4 presents an example where the subscriber never receives messages due to a topic name mismatch. `Conflicting Publishers` misconfigurations appear when there is more than one simultaneous publisher to a topic that should only be accessed by a single publisher. This leads to messages that may provide opposite, conflicting instructions.

The message_filters API is used to filter incoming messages on a given topic (i.e., messages satisfying a given condition trigger a callback) and synchronize messages across multiple topics (e.g., invoke a callback once data is received from multiple sensors). In particular, we noticed difficulties in using this API (`Filters`) to synchronize two topics without losing messages. Finally, `Periodic` misconfigurations occur when the correct system execution relies on messages being periodically published at a specific frequency (e.g., camera, lidar, IMU data). Misconfigurations occur when a component stops publishing data continuously and other parts of the system continue to wait, indefinitely, to receive that data. For instance, a pedestrian detection node must consistently publish images regarding pedestrian's position estimates to function correctly.

Launch. Launch files are the primary means of orchestration within ROS, used to launch and glue together individual components with specific configurations into the system.

Developers often use the ability to import other launch files recursively to improve modularity and simplify reuse by writing individual launch files for separate sub-systems (i.e., a collection of components that work together to perform tasks such as perception, planning, or control). When writing a launch file for an entire system, developers introduce

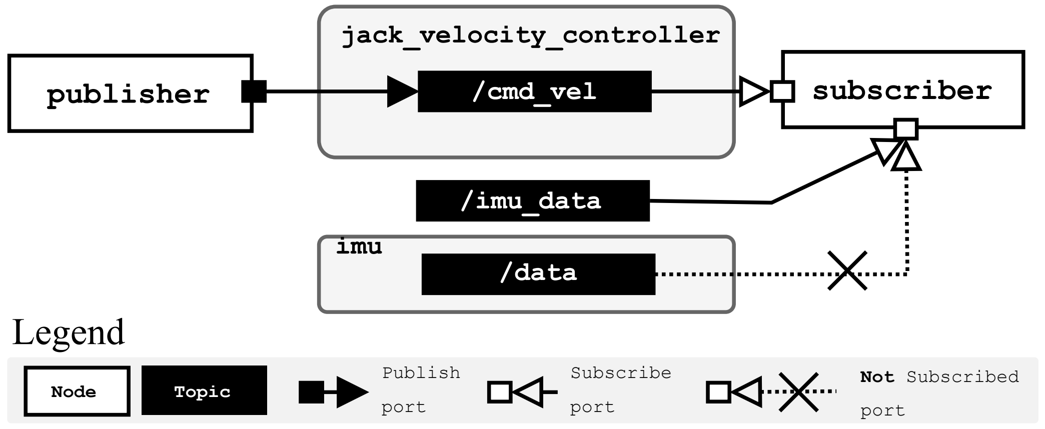


Figure 3.4: Example of a `Name Mismatch` from ROS Answers, where the developer mistyped the subscriber's topic name. The subscriber expects data from `imu/data`, but due to the wrong connection to `imu_data`, no data is received.

a layer of abstraction, requiring them to only reason about what launch files to include rather than worrying about configuring every individual node and subsystem. We observed cases where the developer either `Includes` inappropriate launch files for the given context or fails to include crucial launch files necessary for the robotic system's functioning.

We also observe node `Duplication` errors in launch files, where two or more nodes are instantiated with the same name (e.g., by accidentally launching the same node with the same name). In this case, ROS complains at run-time that the name is taken, and the second node crashes upon launch.

Since architectures in ROS are defined at run-time, launch files are susceptible to `Race Conditions`. For instance, nodes may publish messages to topics before the subscriber finishes launching, causing those messages to be lost. Nodes can also be sensitive to the ordering of `<node>` and `<param>` tags within launch files: In ROS 1, a `<node>` may launch before the parameters are stored on the global parameter server, leading to parameter misconfigurations.

To allow components to be customized to a particular system, launch files support `Arguments`, whose values may be provided by the command line, a parent launch file, or a specified default value. Those can be accessed via string interpolation (e.g., via `$(arg name-of-arg)`) within the launch file or through the command line, and are typically used to specify ROS parameter values, rename nodes, control the inclusion of particular nodes, or set the system time. Arguments are prone to many of the same issues as parameters (e.g., dead writes and unintentional use of default values).

`Environment variables` are also used to customize the behavior of individual components in launch files. Misconfigurations can occur when necessary environmental variables for nodes are not specified or when incorrect values are assigned to those variables.

Parameters. In ROS, parameters are used to adjust the behavior of components to their intended deployment. Parameter values are typically provided by launch files, which are used to compose multiple components into a functioning ensemble. Listing 3.1

```xml
<!-- robot urdf model -->
<param name="robot_description" command="cat $(find urdf_pkg)/urdf/my_robot
    .urdf" />

<!-- robot state publisher node -->
<node pkg="robot_state_publisher"
    type="state_publisher"
    name="robot_state_publisher">

    <param name="~tf_prefix"
            value="robot_name"
            type="str" />
</node>
```

Listing 3.1: Example where `tf_prefix` parameter required for operating with multiple robots is **`Missing`**, leading the system to use the **`Default Parameter`** value.

presents an example of a launch file with two parameter-related misconfigurations where the developer forgets to include a `tf_prefix`. This parameter is critical when working with multi-robot systems, to create separate transform trees for each robot. Since the parameter is not defined, the system, by omission, uses the default value, and a single transform tree is used for all the robot systems.

Since parameters are defined and used at run-time within ROS, components may unexpectedly crash when a required parameter is **`Missing`**, or behave in an unintended manner when the component falls back on a **`Default`** value for a missing parameter. Both of these types of parameter misconfiguration can also be caused by **`Dead Writes`** where the wrong name is used to specify a parameter (e.g., due to a typographical error or a name change refactoring). These cases can be hard to debug as there is no static checking and warnings may not be produced for missing or unused (i.e., dead-write) values.

Misconfigurations can occur when using **`Incorrect`** parameter values. Those values may be universally incorrect (e.g., out of bounds) or contextually incorrect for the given robot, environment, and application. When defining parameter values, developers also need to be cautious of potential **`Dependency`** issues, where the behavior of a given parameter is changed by the value of another parameter (e.g., a parameter that enables or disables a feature).

Semantic Types. Even when components correctly make assumptions about the message format shared on a given topic, they can still make incorrect assumptions about the message content (i.e., their semantic types). For instance, two components may correctly exchange a `sensor_msgs/Image`, but the publisher sends color images while the subscriber expects grayscale images.

Components implicitly assume that messages satisfy specific **`Constraints`** over their contents. For example, values are within certain bounds (e.g., positions, velocities, motor values), specific coordinate frames are used, or the range of laser scan measurements is respected. Type-related misconfigurations may stem from the improper use of images and point clouds, such as mismatched assumptions between components about the **`Color Format`** of the image or point cloud (e.g., grayscale vs. color images) or an incorrect assumption that all of the images and point clouds that are shared on a given topic have been subject to specific **`Pointcloud Transformations`** (e.g., resizing, compression, or color conversion) or **`Image Transformations`**. Finally, misconfigurations can occur due to mismatched assumptions on the **`Physical Units`** of data that are exchanged between components. As robots interact with the real world, ensuring that the component's physical units match is critical. For instance, a publisher describes rotational velocity using radians per second, but the subscriber expects the same quantity in degrees per second.

Names. Every node, topic, service, action server, and parameter within ROS has an associated name specified at run-time either as a field or property within a Launch XML file or as a string in the source code of a component.[2] Since names are resolved at run-time, it is easy to introduce **`Mismatches`** between two or more components in the name of a resource (e.g., topic, parameter, service, or embedded DSL configurations). Figure 3.4 illustrates an example of one of these errors from our dataset, where the developer incorrectly defined the topic's name in the subscriber. Since the name is incorrectly configured, the subscriber receives no messages as there are no publishers for that topic. Detecting this misconfiguration is challenging, as the topic names only differ by a specific character and are usually only detected during execution when the system does not behave as intended. Another instance of a name mismatch relates to typos when writing the configurations for the robot localization. For instance, the developer incorrectly defines the name of the `world_map`, a parameter specifying the frame treated as a fixed reference frame.

ROS implements a hierarchical naming structure to promote encapsulation. By convention, this system groups related resources and allows multiple instances of the same node to be used simultaneously (e.g., one node for each camera). ROS's hierarchical naming structure is implemented via namespaces: Every resource belongs to a namespace, denoted by a forward slash in the name of that resource (e.g.,`/right_camera/raw_image` belongs to `/right_camera`). Furthermore, namespaces may be stacked (e.g., `/vehicle_a/right_camera/raw_image`) where `/` denotes the root of the hierarchy, known as the global namespace. We observe that **`Namespaces`** misconfigurations occur most commonly when developers forget to use namespaces or otherwise use the wrong namespace, leading to naming collisions and causing the system to have an unintended architecture.

In addition to namespaces, ROS also relies on name **`Remapping`** to help encapsulation and reuse: ROS launch files allow the mapping of the name of a certain resource (e.g., topic, service) onto a different name within the context of a particular component. This ability is used to wire a general-purpose component into the necessary configuration for a specific system. For instance, when using the `image_proc/resize` component[3] to resize camera images, users must remap the subscribed `image` topic (i.e., incoming camera images) and published ~`image` topic (i.e., resized image) onto appropriate source and destination topics (e.g., `/right_camera/image_color` and the topic target named `/right_camera/image_color_resized`). Incorrect or missing remappings can lead to message loss and unintended behavior.

[2]http://wiki.ros.org/Names
[3]http://wiki.ros.org/image_proc

Nodes. Nodes (i.e., components) are the processes within ROS that collectively form a working robot by performing computation, exchanging information, and interacting with the robot hardware. In our study, we observe cases where either crucial components are **`Missing`** or an **`Incorrect`** component is used for the particular robot, environment, or intended application. For example, the developer incorrectly uses a planner, which leads the robot to navigate erroneously, possibly against walls. A special case for missing a component is a missing nodelet manager. In this case, developers do not define the qualified manager name, allowing all nodelets to be assigned to that manager.

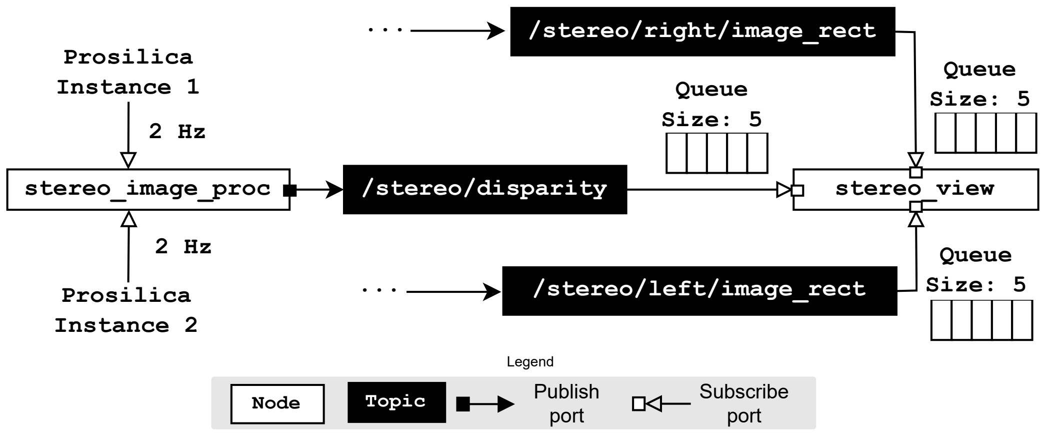


Figure 3.5: Example of a misconfiguration where the developer gets images from `stereo_image_proc` at 2Hz, converting them to disparity images. The `stereo_view` needs matching left, right, and disparity images, but the slow processing speed of disparity images causes left and right images to arrive faster and fill the queue, dropping messages. The `stereo_view` skips these until a matching of images is available.

Timeliness. Robots are real-time systems: messages must be sent between components (e.g., control signals, state estimates) by a certain deadline to prevent unintended and dangerous behavior. Timeliness misconfigurations occur when the timing assumptions of interacting components are mismatched.

Both publishers and subscribers within ROS 1 have associated **`Queues`**, which are used to buffer either outgoing or incoming messages. Determining an appropriate size for this queue is crucial to ensure both timely and correct behavior: A queue size that is too small can lead to message loss, whereas an overly long queue can lead to excessive compute resource usage and message delays. For instance, Figure 3.5 presents a queue misconfiguration within a robot with stereo vision. The `stereo_view` node requires three images: left, right, and the disparity image (i.e., the difference between left and right). In the proposed architecture,[4] the disparity image is computed using the other images. However, computing the disparity image takes time and computational resources. Due to the desynchronization of the sensors, when the disparity image is ready, the original left and right images have already been overwritten in their corresponding queues. To fix this misconfiguration, the developer can

[4]https://answers.ros.org/question/9108

increase the queue size to avoid overwriting or throttle the publishing rate of the cameras to account for time taken to produce the disparity image.

This example is an instance of `Frequency` misconfiguration between publishers and subscribers, where subscribers expect to receive messages at a given frequency to operate correctly.

To function safely, certain components rely on an uninterrupted stream of data that accurately describes the state of the robot and its environment. For example, motion planning in a dynamic environment requires timely and accurate estimates of the position of both the robot and potential obstacles in the scene. Unintended and unsafe behavior can occur when `Stale Data` no longer accurately represent the current state of the robot and its environment, as messages are not published at a high enough frequency.

Finally, `Synchronization` of certain messages is essential for correct system operation. For instance, in example to showing the queue misconfiguration, Figure 3.5 presents an example where the sensors used are sources of multiple misconfigurations. One source of misconfiguration is the lack of synchronization between camera frequencies. There is a mismatch when the stereo_view processor receives the information from each camera. Synchronization of the receiving data is required to ensure that all images are consumed simultaneously, keeping the original images available before consuming the corresponding disparity image.

TF. Robot systems typically rely on a large number of 3D coordinate frames to reason about the relative position and orientation of the robot, its physical parts, and its environment. `tf`[5] is a core ROS library that uses a special tree structure to allow users to transform between coordinate frames (e.g., to determine the position and orientation of the robot's gripper relative to the robot's base).

We observe three major types of TF-related misconfigurations: `Incorrect Transform`, either due to a typo, a misunderstanding of the transform tree semantics, or a mistake about the geometry of the robot. `Missing Transform`, where the developer forgets to provide a transformation between a parent and child frame. Finally, `Duplication` of transforms, where the developer publishes the same TF transform from multiple conflicting sources (i.e., there should be a single source of truth).

Embedded DSL. ROS, its associated toolchain, and some of its most popular (and general purpose) packages rely on their own custom configuration formats. Some ROS packages embed their configuration formats inside of ROS's parameter system, effectively forming an embedded DSL. In other cases, a standalone file is used (e.g., URDF). We observed issues related to the use and configuration of the Navigation Stack, *ros_control*, *robot_localization*, MoveIt!, and URDF. These misconfigurations contain more specific types of issues (e.g., parameter issues). This high-level category presents the types of configuration files related to these misconfigurations.

`Navigation Stack` provides mobile robots with the ability to use odometry and sensor values to localize their position and navigate within a 2D plane by sending velocity commands to the mobile base.[6] Specifically, we saw issues concerning the correct definition of motion

[5]http://wiki.ros.org/tf
[6]https://wiki.ros.org/navigation

planning parameters. For instance, developers may select an inappropriate planner or fail to adapt the parameters of that planner to the robot and environment.

`URDF` (Universal Robot Description Format) files describe robots in terms of their links, joints, transmissions, sensing capabilities, collision geometry, and physical properties (e.g., inertia, contact coefficients, joint dynamics).[7] URDF files are used for visualization, simulation, and motion planning. For instance, when developers forget to specify the joints between two links or incorrectly provide the robot description of two systems in the same file.

`MoveIt` is a platform for building manipulators using ROS that incorporates algorithms for motion planning, manipulation, kinematics, control, and navigation [30]. Successfully configuring it for a robot in a particular environment relies on careful configuration of numerous parameters (e.g., planner density, and padding offsets).

`ROS Control` allows developers to integrate and compose multiple off-the-shelf control algorithms into their system. To behave safely and operate as intended, each controller needs to be adapted to the specifics of each robot, which, when done incorrectly, can result in a misconfiguration.

Finally, `Robot Localization` provides a node collection for performing state estimation (i.e., Kalman filters) and integrating GPS data: they fuse data from multiple sensor sources (e.g., IMUs, GPSs, odometry) to obtain a robust estimate of the robot's state (i.e., position, rotation, velocity). We observe issues that stem from missing or incorrect transforms, incorrect units, and sensor mismatches (e.g., attempting sensor fusion without wheel or visual odometry).

Calibration. Robots rely on a suite of sensors to perceive their environment. To ensure that the robot's understanding of its environment is and remains accurate, those sensors must be calibrated. In our study, we observed misconfigurations due to the miscalibration of `Cameras`, `Odometry`, and `PID` controller parameters, all of which required a manual change. In all of these cases, the mistake was either (a) forgetting to calibrate entirely (e.g., camera intrinsics and extrinsics), (b) relying on default parameters that were inappropriate for the robot (e.g., PID defaults), or (c) using incorrect values (e.g., wheel radius).

Contextual. Misconfigurations can also occur when the system's configuration is tweaked according to a specific context. A simple example is developing the robot's software within a simulation context and deploying it to a physical robot or vice versa. Some configurations and parameters need to be different when changing contexts, leading to `Simulation vs. Real` misconfigurations when the behavior does not match (typically by not adjusting the necessary configurations). Furthermore, we encountered `Application-Environment` misconfigurations where the component's configuration depends on the type of application of the system and the surrounding environment. For instance, the configurations of the components inside and outside a warehouse can be different due to weather, lighting conditions, and surface grip.

The robot's physical configuration (i.e., its sensors, actuators, mechanical components) naturally imposes restrictions over the set of plausibly correct software configurations; configuring the robot's software therefore requires an understanding of the robot's physical

[7]https://wiki.ros.org/urdf

configuration. We identified four sub-categories where `Hardware` components require careful software configuration: (1) `Actuators`, where the configuration of the parameters for a component in the system depends upon the exact actuators that are used, (2) `Sensors`, where the type of sensors and their positioning on the system restricts parameter configuration, (3) `Mechanical`, where the robot's mechanical hardware imposes restrictions of the space of meaningful and safe parameters, and (4) `Compute` where the configuration impacts resource usage. Furthermore, robots may have specific hardware limitations that must be addressed via software configuration. For instance, one source of misconfiguration in Figure 3.5 is resource-related, where components take longer to process overly large camera images.

Other Challenges. During the analysis, we encountered questions not related to misconfigurations. `Documentation` questions occur due to missing or outdated documentation in ROS. This category is not considered a misconfiguration as the developer could not progress with the configuration and reach a misconfiguration due to the lack of documentation. We highlight this category as part of our taxonomy since outdated or lack of documentation may cause developers to introduce misconfigurations. Developers may not know what components to use or how to use them correctly in their systems. We also encountered instances where the issues were related to the component having a `BUG:Component` rather than being a configuration issue, or the `BUG:Infrastructure` in which the system is running contained an error (e.g., RViz). Finally, we identified cases where the misuse of `Simulation` prevents the correct functioning of the system. Developers can use simulation time to playback previously recorded data (e.g., sensor readings) in a time-synchronized manner (e.g., during testing and debugging). However, developers unfamiliar with the concept of simulation time may forget to define it, resulting in incorrect system behavior (e.g., different times in the receiving of data).

3.2 Study of Existing Tools

After categorizing the different types of misconfigurations in ROS-based systems, it is essential to identify which categories are overlooked when using existing state-of-the-art analysis techniques. To that end, we address the following question:

RQ2: To what extent do current techniques address these categories of misconfiguration?

Answering this question reveals the gaps in current tooling and helps guide the development of new tools to increase the coverage of misconfigurations that can be detected earlier in the development process, avoiding errors in deployment. To address this objective, we conducted a literature review following best practices outlined by Snyder et al. [127]. Our search strategy, influenced by Albonico et al. [4], focused on sourcing works from the software engineering, software architecture, software testing, and robotics communities, which are the likely places for such techniques to have been reported. Figure 3.6 illustrates our methodology for performing the literature review, which we describe below.

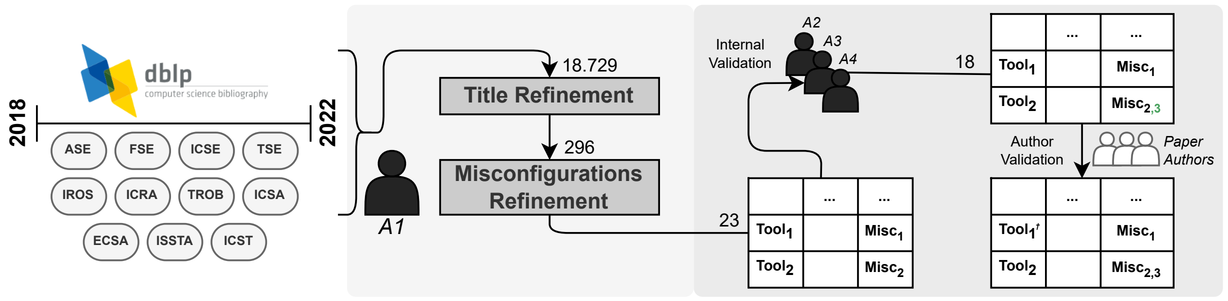


Figure 3.6: Methodology of the literature review. We collected prior work from 2018 to 2022 from top conferences and journals in software engineering, robotics, architecture, and testing. We refined the search by looking at the paper titles, then reading the papers and matching them to misconfigurations. We perform an internal validation with three authors and author validation with the authors of each technique to validate the proposed tool for misconfiguration matching.

3.2.1 Methodology

Our literature review methodology consists of three stages:

Stage 1. Collection: Using DBLP,[8] we collected all papers of journals and conferences and associated co-located events between 2018–2022 inclusive from the major software engineering (ICSE, FSE, ASE, TSE), software testing (ICST, ISSTA), software architecture (ICSA, ECSA), and robotics (ICRA, IROS, TROB) venues. We gathered 18 729 paper links, titles, venues, and respective years.

Stage 2. Refinement & Collection: The first author started by manually inspecting paper titles and searching for keywords that describe any of the subcategories of misconfigurations. Furthermore, for non-robotics venues, we searched in the titles for robotics-related topics. In contrast, in the robotics venues, we searched for concepts related to verification, testing, and repairing misconfigurations. Whenever a prior work seemed relevant, we manually inspected it by looking at the abstract, followed by the introduction, approach, and conclusion.

Then, we annotated the relevant papers with the misconfigurations they cover and their type of analysis (static or dynamic). We consider a technique static if it performs verification without executing the system and dynamic otherwise. Although general-purpose testing techniques potentially cover all the identified misconfigurations in theory, we only considered those that explicitly cover them in their problem statement or examples provided in their evaluation. This refinement considered 18 relevant papers.

Stage 3. Internal & Authors Validation: We performed a two-step validation of the matching each technique and the misconfiguration. We first validated our findings internally with the other authors of this study (A_2, A_3, A_4), experts in software architecture, software engineering, and robotics, to mitigate against missing or incorrect categorizations. We then asked the authors of each technique to validate our findings externally. Since ROS Answers questions do not contain executable examples to test each technique's ability to detect a

[8]https://dblp.org

particular kind of misconfiguration, the external author validation helps us validate our assumptions of each tool's ability. To perform an external author validation, we gathered the contact details for the authors of each technique and emailed them to asking if our classification is correct and what other misconfigurations, if any, their technique addresses.[9] We analyzed the answers received by the papers' authors. When in disagreement, we compare both mappings and re-analyze the paper. From the proposed misconfiguration mapping, 3 out of the 10 authors of each technique proposed an update by adding only 1 extra category. For all but one of thise addition, we agreed with the correction and updated our categorization. The sole case where we disagreed with the authors' additional categories was Santos et al. [125], which allows developers to synthesize run-time monitors and discover bugs by writing properties using the HAROS Property Language (HPL) [124]. Similar to general-purpose test cases, these properties act as tests that may detect specific instances of misconfiguration but do not cover the overall *category* of misconfiguration. While this extensibility is essential, as is writing integration tests in general, we consider this and other similar general-purpose testing techniques outside the scope of this literature review as, with the appropriate instrumentation and scaffolding (i.e., test inputs), testing tools can theoretically identify any misconfiguration, even if it is seldom practical.

3.2.2 Threats to Validity

External Validity. We identify the generalizability to other venues and the time frame selected as external threats. This literature review focused on searching for relevant work at top conferences and journals in software architecture, software engineering, software testing, and robotics. However, our findings may miss relevant tools by focusing on specific conferences and a particular time frame (2018–2022). Extending the search for further years and conferences and journals presents a challenging task as the intersection of multiple research areas quickly increases the prior work required for manual inspection.

Internal Validity. We identify the manual inspection of prior work and the subjectivity in categorization as internal threats. The initial paper refinement was done by only one of the authors, and there is a threat when selecting relevant prior work based on the paper titles where the author may overlook relevant papers. By searching for terms related to misconfigurations and the verification, testing, and repair of robotic systems, we expect to mitigate this threat. There is also a threat when performing the mapping of each technique to a misconfiguration. To mitigate this mapping, each technique was analyzed by two authors, and we tried to validate the mapping with the original paper authors.

3.2.3 Results

Table 3.1 presents the mapping between the techniques and the misconfigurations. We identify the misconfigurations each technique addresses, the type of analysis it performs, static or dynamic, and the venue. We identified techniques that statically or dynamically detect misconfigurations, although some tools are static techniques whose verification is

[9]At the time of submission, 10 of the 18 techniques were validated by the authors of those techniques.

Table 3.1: Overview of each technique, the type of analysis (**D**ynamic or **S**tatic), and the sub-categories of misconfigurations each addresses. The main categories of misconfigurations are as follows: **(Ca)** Calibration, **(Co)** Contextual, **(M)** Messages, **(N)** Names, **(O)** Other, **(P)** Parameter, **(T)** Semantic Types, † Author validated, ∗ Authors Disagreement.

Reference	Venue	Year	Analysis	Misconfigurations
Kate et al. [72]†	FSE	2018	S	**(T)** Physical Units
Burgueño et al. [16]†	RoSE	2018	S/D	**(T)** Physical Units **(T)** Constraints
Witte and Tichy [146]	RoSE	2018	S/D	**(N)** Mismatches **(M)** No Publisher
Wüest et al. [148]	ICRA	2019	D	**(P)** Missing
Cramariuc et al. [31]	ICRA	2020	D	**(Ca)** Camera **(Co)** Sensors
Carvalho et al. [25]†	IROS	2020	D	**(T)** Constraints **(M)** Format **(M)** No Publisher
Wigand et al. [145]	IROS	2020	S	**(Co)** Actuators **(Co)** Application-Environment **(Co)** Mechanical
Kate et al. [73]	FSE	2021	S	**(TF)** Incorrect Transform **(TF)** Missing Transform
Jung et al. [71]†	FSE	2021	D	**(P)** Incorrect **(P)** Dependency **(P)** Defaults **(Co)** Application-Environment
Kortik and Shastha [80]	ICRA	2021	S/D	**(M)** Format **(M)** No Publisher **(M)** Conflicting Publishers
Santos et al. [125]†∗	RoSE	2021	S/D	**(N)** Mismatches **(M)** No Publisher
DeVries et al. [37]†	SEAMS	2021	S/D	**(Co)** Application-Environment **(Co)** Mechanical **(Co)** Sim-Real
Taylor et al. [129]†	ASE	2022	S	**(T)** Physical Units **(TF)** Incorrect Transform **(TF)** Missing Transform
Kim and Kim [75]†	FSE	2022	D	**(DSL)** URDF **(P)** Incorrect **(Co)** Sim vs Real **(Co)** Sensors **(Co)** Actuators **(BUG)** Infrastructure
Das et al. [33]	ICRA	2022	D	**(Co)** Sensors **(Ca)** Camera
Heiden et al. [63]†	ICRA	2022	D	**(O)** Simulation
Timperley et al. [133]†	ICSA	2022	S	**(M)** Conflicting Publishers **(M)** No Publisher **(M)** Format **(N)** Mismatches **(N)** Remapping **(P)** Dead Write **(P)** Incorrect
Han et al. [61]	ICSE	2022	D	**(P)** Incorrect **(P)** Dependency

optionally extended with dynamic analysis. For instance, Burgueno et al. [16] statically analyzes physical units described by a modeling language and generates model invariants checked during the execution of the system.

We also identified techniques that do not detect misconfigurations but rather automatically infer or optimize configuration parameters. The inference of the configurations helps prevent misconfigurations, as developers do not configure the system manually. For example, Wuest et al. [148] automatically infers geometric and inertia parameters, and Heiden et al. [63] probabilistically infers simulation parameters.

For each technique, we describe the misconfigurations it addresses with an important caveat. Although a technique addresses a specific misconfiguration, it does not mean it is solved. Some techniques address particular misconfigurations for specific robotic systems (e.g., Swarmbug) and contain limitations. For instance, Phys [72] and SA4U [129] both detect physical unit misconfigurations. While Phys performs static analysis, allowing it to detect physical unit errors before execution, SA4U requires execution information to detect the misconfigurations.

Overall, we identified 18 related works that address 23 of 50 sub-categories of misconfigurations. Parameters, Messages, and Contextual are the misconfigurations most addressed by current techniques. On the other hand, no misconfigurations related to the Timeliness, Nodes, and Launch categories are currently addressed, and within the Embedded DSL category, only the URDF dialect is considered in current techniques.

3.3 Related Work

In this work, we studied the types of misconfigurations that developers face and what techniques can address them. As the presented misconfigurations are not specific to any ROS version or distribution, we expect our findings to generalize to the many ROS systems.

Prior work studied types of bugs in robotic systems and autonomous vehicles. For instance, 27.25% of the bugs in autonomous vehicles (AV) software detected are misconfigurations [52]. Instances of the misconfigurations we encountered were also found in Unmanned Aerial Vehicles (UAV) [142]. 19.6% of the bugs encountered in a study of two UAVs, PX4 and Ardupilot, are misconfigurations such as parameter misuse and missing, parameter limits related to hardware, and inconsistencies related to sensors and libraries. Our taxonomy of misconfigurations not only addresses ROS-specific issues but also other types of misconfigurations related to the cyber-physical nature of these systems.

Similar to the misconfiguration we encountered, the simulation to real-world transition is challenging in the services robotics domain [53]. Missing dependent components are also described in the literature [47] and presented in our taxonomy through particular instances of missing nodes and missing nodelet managers. Physical unit misconfigurations within ROS have also been quantitatively and qualitatively studied through projects on GitHub [22, 104]. Issues related to URDF files presented in this study, through the scope of Xacro XML language, are also a source of misconfigurations in prior work [2]. ROSDiscover [133] and ROSInfer [39] identify misconfigurations related to the structural composition and behavioral interaction between components, covering certain misconfigurations within our Naming, Parameters and Messages, and Timing categories. Finally, the ROBUST project, [131, 132], a large-scale study of bugs in ROS, evidenced different types of misconfigurations related to missing runtime dependencies (e.g., nodes and configuration files), dangerous defaults (e.g., missing setting a parameter value), namespaces misconfigurations, and name mismatches. Unlike prior work, we focus on how developers misconfigure their systems and which techniques are available to address them.

Configuration errors are not specific to robotic systems, and prior surveys found these in other configurable systems [151]. For instance, some open source storage systems are prone to inconsistency errors of parameter values and value inconsistencies [153], similar to the incorrect parameter and dependent parameter errors we encountered, respectively. Similarly, Android manifests are also a source of incorrect attribute names and values [70]. Finally, configuration errors in databases arise according to the data types used and the ranges of accepted values for the configuration parameters [152].

3.4 Discussion

Misconfigurations are a critical concern in robotic systems, as these lead to unintended and potentially dangerous behavior. Our study and literature review identified a gap in the ability of state-of-the-art analysis tools to cover the space of ROS misconfigurations. In this section, we discuss some of the requirements that future analysis tools need to satisfy to improve the detection of the different categories of misconfigurations.

Misconfiguration analysis must work with ROS's domain-specific languages and dialects. Building ROS Systems requires changing configuration elements distributed in multiple different configuration formats (e.g., Navigation Stack, MoveIt!, URDF). Through our study, we observe that these formats are a source of misconfiguration (**`Embedded DSL`**). As manually tracking many component configurations across different file formats and ensuring their consistency is challenging, automated techniques must detect misconfigurations throughout these files. Through our literature review, we observe that only one technique explicitly treats these DSLs as first-class entities as part of its verification [75].

One avenue to address this concern is considering the DSL configurations in analysis tools. Current analysis tools do not explicitly consider the semantics of the configurations within the different embedded DSLs. A future direction in improving the detection of misconfigurations is to incorporate this knowledge into analysis tools to enhance their capabilities. Alternatively, these separate configuration formats and files could be merged into a single analysis, verifying that configurations are correctly integrated across DSLs.

Misconfiguration analysis require information about the robot's physical environment, hardware, and intended application to reliably detect misconfigurations As cyber-physical systems interact with the real world, correctly defining robot configurations depends on the context in which the system is used. For instance, in Section 3.1.3: **`Contextual`**, we identified misconfigurations arising from the lack of knowledge when changing configurations that depend on the environment, hardware, and type of application. For instance, the positioning of sensors in the hardware, indoor and outdoor environments, the frequency, quality, and size of images provided by the sensors, and the type of robotic system are all factors found in this work that impact the configuration of software components. If analysis tools intend to improve their verification, they must consider this contextual information. However, the physical environment and hardware information is often missing, as our literature review found that only 6 of 18 consider this information to optimize the configuration values.

Future work can provide application, physical environment, and system hardware information to analysis tools to improve their verification through two possible approaches: domain-specific languages and artifact mining. Domain-specific languages have been successfully applied in other domains to verify system properties and improve code quality [74, 81]. Introducing a DSL, allows developers to specify properties regarding the context in which the system is executed (e.g., whether it is executed indoors or outdoors). When existing, contextual information can be obtained by analyzing artifacts (e.g., Phys [72], ROSDiscover [133], SA4U [129]), and inspecting the information these dialects provide.

Static analysis is not sufficient to detect all misconfigurations. Tools must be able to analyze run-time behavior. As executing the system is expensive, time-consuming, and possibly dangerous, it is ideal to detect these misconfigurations prior to system execution using static analysis tools. While static analysis techniques perform great work in reducing the cost of detecting misconfigurations, these are still bounded by the limited context understanding, which is not provided in ROS-based systems, and have scalability issues, being challenging to analyze large codebases [83]. Detecting some misconfigurations requires complex runtime behavior information not available to static analysis tools. For instance, a set of parameter values may need to be corrected according to the system's execution (`Incorrect Parameters`),[10] the incorrect calibration of the system is only detectable when executing the cameras (`Calibration`),[11] or compute issues may arise when hardware actively interacts with the real world (`Contextual`).[12]

Future analysis tools can improve their detection of misconfigurations by augmenting the static checking with properties. Runtime behavior information could be used to generate test cases, monitor runtime configurations, or use machine learning techniques to predict potential misconfigurations based on contextual information and the system's execution. These different approaches for analysis are currently used in techniques observed in our literature review and present a promising research direction. For instance, HAROS [124] generates monitors to track properties during runtime, SA4U [129] instruments the source code to obtain runtime information to help detect physical unit misconfigurations, and Swarmbug [71] performs multiple executions of the system while removing environment configuration variables to detect configurations responsible for the buggy behaviors.

Furthermore, future research can focus on system properties defined using specification languages. Properties in these domain-specific languages can help monitor misconfigurations

[10]https://answers.ros.org/question/30235/
[11]https://answers.ros.org/question/10975/
[12]https://answers.ros.org/question/195186/

that are only detectable dynamically while interacting with the real world.

3.5 Conclusion

In this work, we conduct an empirical study to categorize the misconfigurations that occur within ROS and determine the extent to which existing analysis tools cover those misconfigurations. We find 50 categories of misconfiguration, of which 27 are found not to be addressed and 23 are partially addressed by existing tools. Through this study, we identify promising areas for future research, outline requirements for future analysis tools, and, through our taxonomy, identify where detailed datasets are needed to develop tools to detect specific categories of misconfiguration.

Chapter 4

ROSpec: A Domain-Specific Language for ROS-based Robot Software

Domain-specific languages (DSL) have emerged as a promising approach for describing and specifying user intent over system architectures across various domains, including cloud computing [13], robotics [102], and cyber-physical systems [140]. Within the domain of robotics, DSLs have allowed developers to express and verify correct component interconnections [58, 101, 105, 144], real-time requirements [38, 49, 90, 96, 114], and hardware-software relationships [49, 126, 144]. However, the successful adoption of these languages in specific domains depends on several critical factors [69]. A language must be sufficiently expressive to allow developers to specify their architecture accurately [69], provide domain specialization to facilitate description [147], offer intuitive and comprehensible syntax [97], and maintain an appropriate level of abstraction that aligns with developers' understanding of the system [147]. However, to the best of our knowledge, no prior work has designed such a language that addresses these requirements to detect ROS-based misconfigurations.

In this chapter, we introduce ROSpec, a ROS-tailored domain-specific language for specifying component configuration and integration. Since ROS is the most widely adopted robotics framework, we use it as a proxy for broader component-based robot software development. ROSpec uses domain concepts to specify properties over ROS configurations, ensuring their correct integration.

We design ROSpec by using ROS-related concepts and studying misconfigurations identified in prior empirical work [21]. This allows the language to be expressive through the use of established programming language concepts (e.g., liquid and dependent types [117, 149]) and usable by embedding domain knowledge. ROSpec specifications considers two different stakeholders, addressing the concerns of each of them: component writers who intend to specify their component's semantics, and component integrators who expect the correct configuration and integration of their components. We demonstrate ROSpec's abilities by modeling a medium-sized warehouse robot system and manually studying and describing partial component specifications from a dataset of 182 questions from ROS Answers, a Q&A platform similar to Stack Overflow.

In this work, we make the following contributions:

- The derivation of properties for ROS-based systems from studying prior work (Sec-

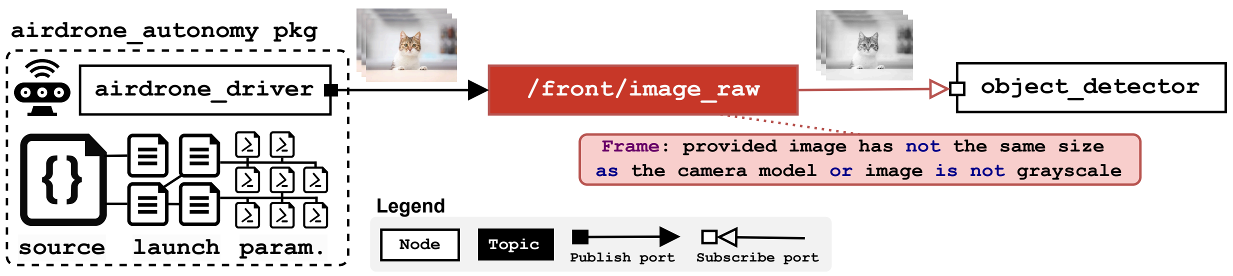


Figure 4.1: Example of a misconfigured publisher-subscriber in ROS.[1] The `airdrone_driver` is composed of multiple source code, launch and parameter files. The subscriber expects grayscale images, but colored (RGB) images are provided. A possible solution is to introduce a node that converts colored images to grayscale.

tion 4.3);

- The introduction of ROSpec, a novel specification language that leverages domain-specific concepts and misconfiguration-related properties (Section 4.4);
- The formal definition of ROSpec's syntax and checking rules that use liquid types to restrict components configurations and their connection integration (Section 4.5);
- The evaluation of the language's expressivity by specifying a medium-sized robot case study (Section 4.6.1) and writing partial specifications of misconfigured components (Section 4.6.2).

4.1 Background and Motivating Example

In this section, we provide an overview of the ROS architecture and community dynamics governing how developers create, configure, and integrate components from its ecosystem. We present the challenges that occur during configuration and integration that lead to misconfigurations.

ROS primarily provides a publish–subscribe architecture based on *nodes* and *topics* for inter-component communication [123]. Nodes process messages received as inputs and may produce new messages to topics — named channels for exchanging messages. ROS also supports other communication models, such as synchronous and asynchronous remote procedure calls (i.e., services and actions). This loosely coupled architecture allows the runtime definition of interfaces, making it easier for developers to create and integrate configurable components into a working system.

Indeed, ROS's key advantage is its rich ecosystem of reusable components, allowing developers to focus on configuration and integration rather than implementing components from scratch. For instance, the Navigation Stack [30] supports autonomous navigation (e.g., planning, motion control, and localization); MoveIt! [27] performs motion planning and manipulation; and ROS Control [28] provides control algorithms and an interface for

[1]https://answers.ros.org/question/201031

interacting with robot actuators. Developers rely on configuration to adapt such components to the requirements of their system.

Within the ROS community, we observe two primary types of developers involved in component creation and configuration: component `Writer` and `Integrator`. A component writer creates reusable packages composed of multiple nodes, focusing on having generic components that allow easy configuration by integrators. These developers contribute their components to the community, expecting its correct configuration and integration. A component integrator selects and adapts these components to meet specific system requirements while ensuring that configurations match the assumptions about their correct usage.

To develop their systems, ROS provides a C/C++ and Python API that abstracts low-level inter-component communication details. Here, developers can define what components are used and configure them using configuration files. Component configuration involves a combination of launch and parameter configuration files, where developers define parameter values like topic names, robot's physical configurations (e.g., URDF), and algorithm configurations [93]. These configurations often depend on the robot's operating environment, hardware setup, and task. However, due to multiple layers of indirection, developers often have to manually search source code and dozens of launch and parameter files to understand which configurations are relevant and how the component integrates into the system's overall architecture. For instance, Autoware.AI,[2] one of the largest and most complex ROS autonomous systems, contains 230 components.

Correctly configuring and integrating these components is critical, as configuration errors, or *misconfigurations* [2, 20, 104, 132], may potentially result in dangerous behaviors of the system when interacting with its environment. Misconfigurations arise from semantic mismatches in components' usage, which are often detected late in the development pipeline, during field testing, or even deployment [1]. For instance, Figure 4.1 presents a structural architectural view of the integration of two components. The `airdrone_driver` publishes colored camera images to `"/front/image_raw"`, while the `object_detector` subscribes to and processes image messages from this topic, allowing the system to avoid obstacles. However, the integration of both components leads to a color-format misconfiguration, where the subscriber assumes all images received are in grayscale while the publisher provides colored images, removing the drone's ability to avoid obstacles.

While ROS's promise of reusability through architectural decoupling may accelerate prototyping, the tradeoff is that integrators are responsible for properly managing, configuring, and integrating dozens, if not hundreds, of components whose documentation is often missing [1, 42, 134], or when existing, is not enforced. This requires integrators to have a deep understanding of components, read the documentation (when available), and carefully provide configurations to have a working system, hampering this promise of reusability. ROSpec addresses these challenges by providing writers a language to specify the semantics of their components and integrators to instantiate them, ensuring the correct configuration and integration of components.

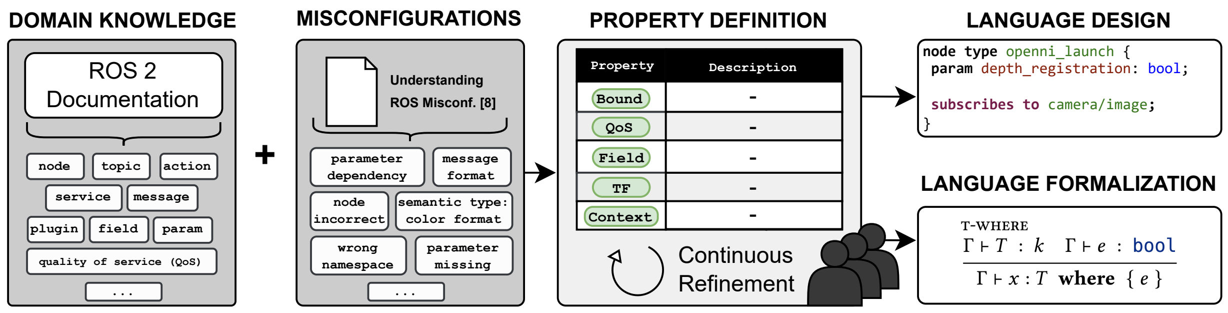
[2]https://github.com/autowarefoundation/autoware

4.2 Related Work

Prior work presented domain-specific and architectural domain languages to specify components and prevent architectural issues in the robotics and cyber-physical domains [102, 140]. At a high level, they focus on specifying structural, behavioral, and hardware-specific domain-based architectural properties. Structural specification languages describe the interconnection between components and connectors [58, 101, 105, 144], ensuring that software components are correctly assembled. Behaviorally, these languages specify properties to ensure real-time (e.g., frequency [49], queues [11], and synchronization [38]), timing [90, 114], and other functional [58, 96] and non-functional [91, 115] requirements. Domain architectural languages also specify hardware properties [49, 126, 144] to ensure the system's software-to-hardware architectural consistency. Finally, synchronous programming languages such as Lustre [60], Esterel [14], and coordination languages like Lingua Franca [89] and Kahn Process Networks [56] provide formal temporal semantics to verify causal relationships and timing properties between components that ROSpec currently does not support.

General-purpose architectural description languages (ADL), such as Wright [5], Architecture Analysis and Design Language (AADL) [45], and Acme [54], have previously been used to describe robot architectures [18, 120]. Specifically to ROS, recent work has described component connections using AADL [11]. However, its application in other domains identified usability challenges, including property ambiguity regarding system requirements and subcomponent specifications (which ROSpec addresses directly) [36], increased entry barriers for adoption [36], and lack of flexibility in supporting user-defined connectors [107]. When specifically comparing AADL to DSLs, such as ROSpec, prior work identifies three main challenges [23]: (1) AADL lacks stakeholder-specific language specialization, making it difficult to separate concerns for different roles; (2) AADL's general-purpose constructs create verbosity and potential conflicts with domain-specific concepts (e.g., *process* and *subcomponent* have different meanings in ROS); and (3) effectively detecting the misconfigurations given Table 4.2 properties requires AADL to model concepts like liquid and dependent types — making it challenging to balance generality and domain specialization.

Recent advances in automated recovery of ROS configurations and architecture have addressed particular instances of ROS misconfigurations. For instance, HAROS [125] and ROSDiscover [133], recover the structural architecture of the system and can detect common connection issues (e.g., subscriber missing a publisher), ROSInfer [40] infers reactive, periodic, and state-dependent information from components, allowing them to detect three behavioral architectural misconfigurations, and Phys [72] detects physical unit mismatches in components by using ROS message assumptions and physical units inference. Nevertheless, the effectiveness of these approaches depends on the user intent. ROS components are very versatile, and the properties of their usage depend from component to component. By using standard assumptions about components, these tools cannot capture the specific configurations and properties of each component, leading to false positive detections. ROSpec approaches this concern by providing a way to specify user intent, which can potentially be integrated with these approaches to detect misconfigurations in the source code.



```
node type openni_launch {
 param depth_registration: bool;

 subscribes to camera/image;
}
```

$$\frac{\Gamma \vdash T : k \quad \Gamma \vdash e : \texttt{bool}}{\Gamma \vdash x : T\ \textbf{where}\ \{\, e \,\}}$$

Figure 4.2: Methodology for designing ROSpec based on ROS concepts and misconfiguration properties for ROS-based robotic systems. We begin by collecting ROS domain knowledge and information on domain knowledge and misconfigurations, using prior work in misconfigurations [21]. Then, we derive and iteratively refine properties from the collected knowledge. Finally, we design and formalize the specification language.

Liquid types [117] have traditionally been used in executable languages, such as Haskell [138], TypeScript [139], Java [50], and Flux [85] to ensure program correctness, verify resource consumption [76], and synthesize programs given a specification [48, 109, 135, 143]. While recent work has explored applying refinement types to non-executable languages for visualization synthesis [150] and security in web applications [84], their application to architectural specification domains remains largely unexplored to the best of our knowledge. As demonstrated through ROSpec, liquid types have the potential to refine configurations and integration in highly configurable systems across domains.

4.3 Language Properties

To address these challenges, we design ROSpec by leveraging core ROS concepts, allowing developers to specify familiar domain-specific concepts. Additionally, we derive and incorporate properties from known sources of misconfigurations into the language, allowing developers to specify constraints over their components. In this section, we present our methodology for collecting relevant ROS domain concepts and deriving properties, informed by a prior empirical study in misconfigurations, and its respective threats to validity.

4.3.1 Methodology

Our methodology adapts a prior work's approach for language design that identifies properties from the analysis of developer errors [32]. We begin by collecting ROS domain knowledge and categories of misconfigurations. Then, we express the properties that model the collected knowledge. Figure 4.2 outlines our methodology for collecting ROS concepts and defining properties components must hold for their correct configuration and integration. We describe each step as follows.

Table 4.1: Overview of ROS 2 main domain concepts, including their names and descriptions, used in ROSpec to describe the configuration and integration of ROS components.

Domain Concept	Description
`Node`	Process that may receive an input, process information and produce an output. Nodes use plugins, and contain required & optional arguments and parameters.
`Topic`	Named buses over which components exchange messages. Nodes communicate with each other by sending and receiving messages from topics.
`Publisher-Subscriber`	A decoupled communication where nodes publish and receive messages from topics. Publisher-subscriber connections are defined when, at the source-code level, there is a creation of a publisher or subscriber from a node to a topic.
`Service & Action`	Client-server communication model based on requests and responses. Components send a request message to another component that processes it and provides a response. Services provide synchronous communication, while actions are asynchronous and provide feedback and task preemption.
`Parameter`	Named configuration values used by nodes to change components behavior at runtime. Parameters can be declared and accessed dynamically.
`Argument`	Command-line inputs provided when launching components (e.g., nodes) or executing ROS 2 commands. Arguments allow developers to dynamically configure nodes by changing default values, selecting parameter or launch files to load, or controlling conditional logic, such as whether to include specific components.
`Messages`	Define the structure of data containing typed fields used in topics. Services and action message fields contain request and response, and also feedback fields.
`Plugin`	Dynamically loaded and used by nodes to extend application behavior without requiring application source code. These provide the same functionality as nodes, containing arguments, parameters, connections, and frames.
`TF Frames`	Coordinate reference frames used to represent positions and orientations of robot parts or environment. These are required for maintaining spatial relationships over time, allowing components to interpret data in a shared coordinate system.
`TF Broadcast/Listen`	Transform broadcast sends the relative position and orientation between two frames. Components may subscribe to broadcasted transforms in the system and save these frames positions over time.
`Remapping`	Runtime change of nodes, arguments, parameters, and topics names without code changes, allowing component reuse in different contexts.
`Quality of Service`	Policies that define how nodes transmit data over topics, including reliability, durability, and latency. QoS settings must be compatible between publishers and subscribers to ensure correct communication.

We started by identifying the core concepts defined in the documentation of ROS 2, the current version of ROS, as ROS 1 reached end of life. For the remainder of this paper, when using ROS we refer to ROS 2. We manually collected the primary concepts from an official page with ROS concepts [93],[3] providing a glossary of domain concepts and their respective description.

The second source of knowledge comes from a prior empirical study on ROS misconfigurations [21]. By collecting and analyzing this study, we design the language to express properties able of detecting real-world misconfigurations. We review the work on ROS misconfigurations [21], which categorizes 12 high-level categories and 50 sub-categories of misconfigurations encountered by developers when configuring ROS systems in ROS Answers, a StackOverflow for ROS questions. At a high level, their study identified cyber-physical misconfigurations (e.g., hardware and operating environment), architectural issues related to components integration, communication timing assumptions, and configuration issues in launch and parameter files. As ROS and other component-based robotics frameworks share these overall concepts, we expect our findings to generalize beyond ROS, though further study is needed.

Domain Knowledge and Misconfigurations. Table 4.1 lists 12 core ROS domain concepts and their descriptions. We focus on the concepts most relevant to developers [123], such as nodes, topics, and publisher-subscriber. This allows ROSpec to specify standard yet core ROS elements while future extensions can support more complex or uncommon concepts.

Overall, the concepts can be grouped into four categories: (i) components and their configurable information and dependencies (`Node`, `Parameter`, `Argument`, and `Plugin`), (ii) communication models with respective topics, messages, and settings (`Publisher-Subscriber`, `Services`, `Topic`, `Message`, `Quality of Service`, and `Actions`), (iii) `TF Frames`, `TF Broadcast`, and `TF Listen`, and (iv) `Remapping`, which allows renaming and reusing components in their system.

Properties Definition.

Table 4.2: Overview of configuration and integration properties, including their identifiers and descriptions, used to detect misconfigurations in ROS-based systems.

Property	Description
`Typing`	The provided argument and parameter values respect the configuration type.
`Bounds`	The provided values for arguments and parameters respect the bounds in types expected in the configuration documentation.
`Dependency`	The value dependencies between arguments, parameters, context and publisher-subscriber connections respect the expected configuration documentation.

Continued on next page

[3]https://docs.ros.org/en/jazzy/index.html

Table 4.2 – continued from previous page

Property	Description
`Presence`	The arguments and parameters required in the configuration definition are provided.
`Consistency`	The arguments and parameters defined in the configuration exist in the set of arguments and parameters expected.
`Conditionals`	The definition of specific arguments and parameters values depends on the existence of a value definition of other configurations.
`Connection`	The components who subscribe/consume information from topics, expect a publisher/producer providing messages to that topic.
`Message`	The message types provided by publishers and servers are equivalent to the ones expected by subscribers and clients.
`Fields`	The content of messages fields provided by publishers and servers is less restrictive than the one expected by subscribers and clients.
`Cardinality`	The number of publishers/providers and subscribers/consumers for a connection to a topic respects the expected in the configuration.
`Context`	The context of deployment scenario where the component executes matches the expected in the documentation.
`TF-Listen`	Components listening to a TF transform expects a respective broadcasting transform.
`TF-Graph`	For all TF transforms by a component with a parent to a child, each child frame expects to contain one parent frame.
`QoS`	The publisher-subscriber, service and action connections to a topic respect the Quality of Service (QoS) settings defined in the documentation.

Table 4.2 presents a set of 14 configuration and integration properties that cause misconfigurations whenever violated in ROS-based systems. Each property describes a specific condition that must hold for a configuration, ensuring the correct configuration.

Properties are natural language specifications that describe assumptions about component configurations and their integration. We manually defined a property for each misconfiguration, considering the domain concepts collected in the prior stage, to address each high-level category and sub-category of misconfiguration. Then, we evaluate the existing set of properties and either create new properties or refine existing ones. As the categories of misconfigurations are not mutually exclusive, the continuous refinement ensured the collection of a minimal yet comprehensive set of properties. The refinement process involved meetings among three authors, experts in robotics, software architecture, and programming languages, to ensure that these represent relevant properties that capture the domain's architectural nature. This process resulted in a set of properties that captures both domain concepts and categories of misconfigurations.

4.3.2 Threats to Validity

External Validity. Our work uses ROS 2 as a proxy for general-purpose component-based robotic systems. While ROS 2 is widely adopted in both academia and industry, the specific domain concepts and misconfigurations identified may not generalize to other robotic middleware or architectures (e.g., YARP, OROCOS). Nonetheless, our methodology is designed to be adaptable: the language and properties can be instantiated over different domain models by redefining the relevant concepts and misconfiguration categories. Future work is needed to validate this generalizability.

Internal Validity. We identify three main threats to the internal validity of our methodology. First, a single author collected the initial set of domain concepts and categories of misconfiguration, which may miss relevant information. We mitigated this with the collaboratively review and refinement with the other three co-authors. Second, the analysis of misconfigurations relies primarily on a single empirical study, which, although comprehensive, may not capture the deeper details for each specific category of misconfiguration identified in other works. Third, the ROS Answers dataset used in prior work does not distinguish between ROS 1 and ROS 2, potentially introducing inconsistencies in how misconfigurations are mapped to the ROS 2 domain. However, as the architectural differences between ROS 1 and ROS 2 are minimal, we believe our properties generalize to both versions.

Construct Validity. The primary threat to construct validity regards the definition of ROS architectural concepts. For example, a publisher connection may be interpreted either as a connection between a node and a topic at the source-code level or as only when a message is published at runtime. In ROSpec, since the goal is to describe the configuration and integration of components from a static architectural view, we adopt the former interpretation, based on prior work [133].

4.4 ROSpec Design

Based on the domain concepts and the misconfiguration properties, we now present the syntax of ROSpec, using examples derived from real-world misconfigurations documented in ROS Answers[4] (now migrated to Robotics Stack Exchange),[5] a Q&A platform similar to StackOverflow for ROS developers. For each specification example, we identify the primary stakeholder responsible for defining the specification (`Writer` or `Integrator`), the main ROS concepts used (`Concept`), and the corresponding misconfiguration property (`Property`).

4.4.1 Node definitions, Parameter Refinements and Dependencies

In ROS, nodes are the first-class elements that receive, process, and send information. Nodes are responsible for starting communication channels, such as publishing and subscribing to topics, and providing or consuming services or actions. Nodes can be dynamically parameterized with arguments (during system execution) or in parameter files. As both arguments and parameters lead to misconfigurations, we support their specification within a node.

[4]http://answers.ros.org
[5]https://robotics.stackexchange.com
[6]https://answers.ros.org/question/364801

```
node type move_group_type {
    param elbow_joint/max_acceleration: double where {_ >= 0};
    param elbow_joint/min_velocity: double;

    optional param elbow_joint/max_velocity: double = 1.2211;
    optional param elbow_joint/has_velocity_limits: bool = false;
    optional param elbow_joint/has_acceleration_limits: bool = false;
} where {
    exists(elbow_joint/max_velocity) ->
            exists(elbow_joint/has_velocity_limits);
}
```

Listing 4.1: `Writer` specification of the `move_group` node, from MoveIt! [27], responsible for motion planning, kinematics, and execution of robotic manipulators trajectories (`Node` , `Parameter`).

```
system {
    node instance move_group: move_group_type {
        param elbow_joint/max_acceleration = 0.0;
        param elbow_joint/min_velocity = 0.0;
        param elbow_joint/max_velocity = 3.14;
    }
}
```

Listing 4.2: `Integrator` configuration of the `move_group` node. The parameter `has_velocity_limits` is defined in the node type as `false`, which is incompatible with the definition of the `max_velocity` parameter. Without detection, the default value is used instead of 3.14. (`Typing` , `Bounds` , `Dependency` , `Presence`).[6]

Listing 4.1 presents an example of a partial specification for a node type, `move_group_type`, created from a question with a real-world issue asked on ROS Answers. The `move_group_type` node type, part of MoveIt!, is responsible for motion planning and trajectory execution for robotic manipulators. A `Writer` can specify their components by describing their configurable information (i.e., parameters, arguments, and contextual information (Section 4.4.5)), their connections (e.g., publisher-subscriber) and conditions that define dependencies between the defined information.

Parameters can be specified as required or optional with a default value, as in ROS, and are typed. Required parameters, such as `elbow_joint/max_acceleration`, must be provided when creating a node instance. Optional parameters, such as the parameter `elbow_joint/has_velocity_limits`, include default values (`false`), which are used when not defined. Optional parameters not defined during instantiation, are inherited by the instance. Parameters have a name, which may include a namespace, i.e., a prefix that groups nodes, topics, and parameters, preventing naming conflicts.

In ROSpec, `Writer` can encode semantic information regarding their parameters through the use of liquid types [50, 85, 117, 138]. Liquid types are a refinement type system that combines type inference with logical predicates to allow automated verification of program properties. For example, elbow_joint/max_acceleration is defined as a double whose value is non-negative, otherwise the elbow joint could start moving backward while performing an operation, when it is not supposed to. Developers often miss these bounds since they are hidden in the documentation or transmitted through informal shared knowledge. By embedding liquid types in the language, we can specify these types of constraints to detect misconfigurations related to incorrect parameter values.

Information dependency is also supported in refinements through the use of logical expressions to relate parameter, arguments, and their presence with each other. In the move_group_type definition, we specify that if a non-default elbow_joint/max_velocity exists in a configuration, then the parameter elbow_joint/has_velocity_limits must also exist. This dependency is a documented requirement that velocity limits must be used when specifying velocity parameters.

Listing 4.2 presents an example of a parameter dependency misconfiguration from an example from ROS Answers [21]. In this case, the Integrator defined the **system**, which uses the specifications from all its components, such as Listing 4.1. Since these files often contain dozens, if not hundreds, of configurable parameters, they may skip the definition of optional parameters that may actually be required due to parameter dependency. For example, the max_velocity is specified (with value 3.14), but has_velocity_limits is explicitly set to false. This violates the dependency constraint defined in Listing 4.1 as it requires velocity limits to be enabled when max_velocity is provided.

4.4.2 Messages Alias and Field Definition

```
type alias ImageEncoding16: Enum[RGB16, RGBA16, BGR16, BGRA16, MONO16,
             16UC1, 16UC2, 16UC3, 16UC4, 16SC1, 16SC2, 16SC3, 16SC4,
             BAYER_RGGB16, BAYER_BGGR16, BAYER_GBRG16, BAYER_GRBG16];
type alias ImageEncoding32: Enum[32SC1, 32SC2, 32SC3, 32SC4,
                                 32FC1, 32FC2, 32FC3, 32FC4];
type alias Meter: int8;
type alias Millimeter: int8;

message alias ImageWith16Encoding: sensor_msgs/Image {
    field header: Header;
    field encoding: ImageEncoding16;
    field data: Millimeter[];
}

message alias ImageWith32Encoding: sensor_msgs/Image {
    field header: Header;
    field encoding: ImageEncoding32;
    field data: Meter[];
}
```

Listing 4.3: Writer specification of message and type aliases to provide documentation regarding image encodings and respective field information often missing (Message).[7]

ROSpec allows the `Writer` to encode domain knowledge about ROS into the types through type aliases. For instance, as presented in Listing 4.3, developers can define a `type alias Meter: int8`, representing a physical unit measurement in meters. These are useful for documentation purposes, since developers understand the semantics behind parameters, while enforcing constraints structurally over the type to avoid misconfigurations. For instance, comparison with different units is incorrect and may lead to physical unit mismatches (e.g., comparing meters and millimeters) [22].

ROSpec also allows the definition of message aliases and their field specifications. For example, Listing 4.3 presents two `message alias` of different types of image encoding from a ROS Answers question. Here, the `Integrator` is unaware of the encoding types, and their relation with the `data` physical unit. The specification of both message aliases ensures that (1) a correct encoding is always used in the encoding field, i.e., since in ROS, these fields are strings, they are prone to typos; (2) the dependency between the `data` physical unit and the `encoding` type is respected and documented; and (3) two components with a connection to the same topic have matching image encodings.

4.4.3 Publisher & Subscriber, Service and Action Connections

```
node type hector_object_tracker_type {
   param distance_to_obstacle_service: string;

   subscribes to worldmodel/image_percept:
                            hector_worldmodel_msgs/ImagePercept;
   publishes to visualization_marker: visualization_msgs/Marker;
   consumes service content(distance_to_obstacle_service):
                            hector_nav_msgs/GetDistanceToObstacle;
}

node type hector_map_server_type {
   provides service /hector_map_server/get_distance_to_obstacle:
                            hector_nav_msgs/GetDistanceToObstacle;
}
```

Listing 4.4: `Writer` specification of the `hector_object_tracker` and `hector_map_server` nodes, responsible for tracking objects in an environment and fusing perceptual data (`Topic`, `Messages`, `Publisher-Subscriber`, `Service & Action`).

```
system {
   node instance hector_object_tracker: hector_object_tracker_type {
      param distance_to_obstacle_service = "get_distance_to_obstacle";
   }
   node instance hector_map_server: hector_map_server_type {
+     remap /hector_map_server/get_distance_to_obstacle to
```

[7] https://answers.ros.org/question/209450

```
+               get_distance_to_obstacle;
    }
}
```

Listing 4.5: Integrator configuration of two nodes from ROS Answers.[8] A misconfiguration is raised as no publisher is provided to `worldmodel/image_percept`, and due to a missing remap, no services are available for `hector_object_tracker` (Connection, Messages).

In ROSpec, a Writer can specify which topics a given node **publishes/subscribes to**, or what services and actions that are provided or consumed. This information is often scattered in the source code since topic names may be defined in parameter files and remapped. ROSpec provides this spread-out information in one place, allowing for a unified view of the architecture.

In Listing 4.4, a Writer defines the partial specification for two node types with connections, `hector_object_tracker_type` and `hector_map_server_type`, responsible for object tracking and map management, respectively. Line 6 uses a special function internal to ROSpec, **`content`**; upon instantiation, the parameter's content replaces the named topic for the service.

Listing 4.5 shows the description of a system containing two misconfigurations. Two node instances are created, inheriting all connections from the respective node types (from Listing 4.4). During node instantiation, they provide the topic name through the `distance_to_obstacle_service` parameter, which replaces the occurrence of the parameter by the service topic name in (Line 6). However, since the topic name does not match the one provided by the service in the `hector_object_tracker` node instance, there is no provider to the service, leading to a missing service provider misconfiguration. Lines 19 and 20 propose a fix to the misconfiguration by introducing a remap that replaces the wrong topic name with the correct one (`get_distance_to_obstacle`). The second misconfiguration occurs since the `worldmodel/image_percept` topic is being subscribed to, but no publisher exists.

4.4.4 Quality of Service (QoS) and Color Format Policies

In ROS 2, Quality of Service (QoS) policies are used to control communication reliability, latency, and resource usage for topics and services. It allows developers to have fine-tuned control over message delivery, supporting features like reliability (best-effort vs. reliable), durability (volatile vs. transient local), history (keep last vs. keep all), deadline constraints, and liveliness checks. In ROSpec, we model these in the form of a policy. *Policies* are tags that decorate an attached structure. Policies have a structure with settings that are used for the verification. The verification of QoS settings follows the rules in the official documentation.[10]

[8] https://answers.ros.org/question/164526
[9] https://answers.ros.org/question/142456
[10] https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Quality-of-Service-Settings.html

```
policy instance best_effort_qos5: qos {
    setting depth = 5;
    setting reliability = BestEffort;
}

node type openni_camera_driver_depth_type {
    optional param depth_registration: bool = true;
    @qos{best_effort_qos5}
    @color_format{Grayscale}
    publishes to /camera/rgb/image_raw: sensor_msgs/Image;
}

node type custom_node_type {
    @qos{reliable_qos5}
    @color_format{RGB8}
    subscribes to /camera/rgb/image_raw: sensor_msgs/Image;
}
```

Listing 4.6: `Writer` adapted specification of openni and custom node with connections and two policies from a ROS Answers question (`Publisher-Subscriber` , `Quality of Service`).[9]

```
system {
    node instance custom_node: custom_node_type {}
    node instance openni_camera_driver: openni_camera_driver_depth_type {}
}
```

Listing 4.7: `Integrator` configuration of both nodes from ROS Answers. The integration contains a color format misconfiguration, since publisher and subscriber contains different image types. We purposedly introduced a QoS misconfiguration leading to QoS mismatches. (`Connection` , `QoS`).

Listing 4.6 presents an example of two node types, `openni_cammera_driver_depth` and a `custom_node`, and their respective QoS settings. `Writer` creates a policy instance by providing the parameters, and attaches it to both publisher and subscriber. Listing 4.7 considers the instantiation of both node instances in the system. In this case, the integration of both components leads to two misconfigurations: (1) the subscriber is expecting `RGB8` image messages, while the publisher is sending `GrayScale` images, leading to color format misconfigurations; and (2) the QoS settings between publisher and subscriber do not match, since the publisher makes a best effort in delivering messages, while the subscriber expects to receive every message.

4.4.5 TF Transforms and Contextual Information

```
 1  node type laser_scan_matcher_type {
 2    context is_simulation: bool;
 3    // multiple parameters here...
 4    optional param use_sim_time: bool = false;
 5
 6    broadcast world to base_link;
 7    listens base_link to laser;
 8  } where {
 9    is_simulation -> use_sim_time;
10  }
```

Listing (4.8) Writer specification of the node type `laser_scan_matcher` containing frame broadcast and listens, and execution context information (TF Frames , TF Broadcast/Listen).

```
11  system {
12    node instance laser_scan_matcher:
13                  laser_scan_matcher_type {
14        context is_simulation = true;
15    }
16  }
```

Listing (4.9) Integrator configuration leading to context and frame broadcast errors (Context , TF-Listen , TF-Graph).[11]

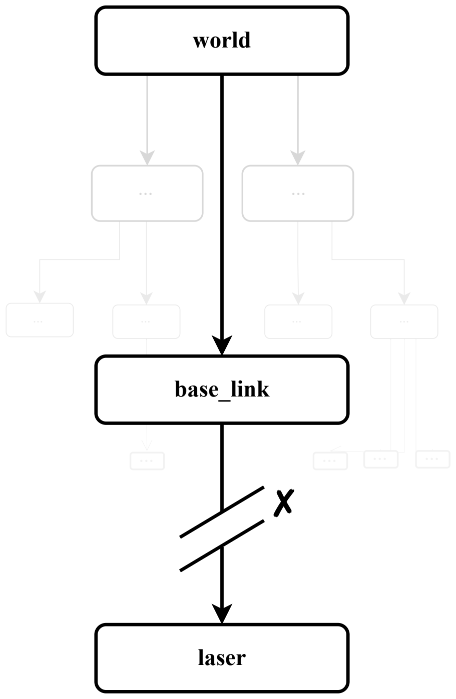


Figure 4.3: Faulty branch of TF tree due to missing broadcast from `base_link` to `laser`.

In ROS, TF frames correspond to coordinate systems attached to parts of a robot or its environment (e.g., base, camera, gripper). tf2 manages the spatial relationships between these frames by constructing a tree of transforms, where each transform defines how to translate and rotate one frame relative to another. This allows any node to convert positions, orientations, or motions from one frame to another, allowing them to perform navigation, perception, and manipulation.

Listing 4.8 presents a partial specification for a `laser_scan_matcher_type` node, which processes laser scan data to estimate robot motion. The node broadcasts a transform from `world` to `base_link` and listens for transforms from `base_link` to `laser` (DP13). The specification also includes contextual information through the `is_simulation` variable (DP16).

A Writer can specify contextual requirements that all instanced system components must ensure. For instance, nodes may have distinct distribution contextual requirements, making their integration impossible. In line 2, the contextual information specifies whether the node is executed in a simulated environment. If so, there is a dependency between `is_simulation` and the `use_sim_time` parameter (Line 9), as it must be `true` to synchronize

[11]https://answers.ros.org/question/11095

with simulated time instead of system time.

In Listing 4.9, the Integrator creates a `laser_scan_matcher` node instance and sets `is_simulation` to `true`. However, this instantiation contains two misconfigurations: (1) as `is_simulation` is true, `use_sim_time` should also be true, but it has its default value of false. Without this verification, the node uses system time rather than simulation time, causing timing inconsistencies; and (2) the specification requires a transform from `base_link` to `laser`, but the integrator does not provide this transform. In a ROS system, this results in the TF system being unable to resolve the relationship between these frames, causing sensor data processing failures. Figure 4.3 illustrates the resulting faulty TF tree, where the transform from `base_link` to `laser` (represented by a dotted line) is defined as required in the specification but not provided in the actual system at runtime.

4.4.6 Plugins

```
node type arm_kinematics_constraint_aware_type {
   param group: Plugin;
}

plugin type right_arm_type {
   param tip_name: string;
   param root_name: string;
   optional param robot_description: string = "robot_description";
   optional param tf_safety_timeout: Second = 0.0;
   // more parameters and connections
}
```

Listing 4.10: Writer perspective of a node and plugin definition from a ROS Answers question (Plugin).[12]

Plugins extend the functionality of node processes at runtime. These contain the same information as nodes: arguments, parameters, contextual information, and connections (e.g., publisher-subscriber). Listing 4.10 shows an example of a plugin type, `right_arm_type`, a kinematics solver responsible for computing inverse kinematics for the robot's right arm. When creating the respective **`plugin instance`**, developers provide the parameters, and *must* assign the plugin instance to a **`node instance`** parameter. A plugin instance is created only when it is used by a node instance. Otherwise, any connections provided do not exist. By designing ROSpec considering plugins, we allow developers to quickly change a node instance configuration by plugging in different plugins.

Overall, ROSpec allows developers to specify and integrate ROS components into a system. In this section, we illustrated the language features from the perspective of two stakeholders. We presented the language syntax and expressiveness by specifying component configurations and their integration in real-world examples. The following section showcases the language verification.

[12] https://answers.ros.org/question/364801

4.5 Language Semantics

In this section, we present the grammar, type formation rules, and type-checking rules used to detect misconfiguration issues that arise when defining and integrating different components in a ROS project. For brevity, we present only the particular subset of the rules relevant to our domain. The complete syntax, formation, and verification rules is provided in the Supplemental Material.

4.5.1 Grammar

$$
\begin{array}{rrl}
\text{Types} & T ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{string} \mid \texttt{t} \\
& & \mid \textbf{struct}\ \{\ \overline{x : T}\ \} \\
& & \mid x : T\ \ \textbf{where}\ \ \{\ e\ \} \\
& & \mid \texttt{Optional}(T,\ e) \\
& & \mid \texttt{NodeT}(\overline{S_p}; \overline{S_c}; \overline{S_{f_r}}) \\
\text{Definitions} & D ::= & \texttt{node type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \} \\
& & \mid \texttt{node type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \}\ \ \textbf{where}\ \ \{\ e\ \} \\
& & \mid \texttt{node instance}\ \ x_1 :\ x_2\ \{\ \overline{S_{p_i}};\ \overline{S_r}\ \} \\
& & \mid \texttt{system}\{\ \overline{D}\ \} \\
\text{Declarations} & S_{p_d} ::= & \texttt{param}\ \ x :\ T; \\
& & \mid \texttt{optional param}\ \ x :\ T\ =\ e; \\
& S_{p_i} ::= & \texttt{param}\ \ x\ =\ e; \\
& S_c ::= & x_1\ \ \texttt{publishes/subscribes to}\ \ x_2 :\ T; \\
& & \mid x_1\ \ \texttt{publishes/subscribes to}\ \ x_2 :\ T\ \ \textbf{with qos}(e); \\
& S_{f_r} ::= & x_1\ \ \texttt{broadcast/listens}\ \ x_2\ \ \texttt{to}\ \ x_3; \\
& S_r ::= & x_1\ \ \texttt{remap}\ \ x_2\ \ \texttt{to}\ \ x_3; \\
\text{Context} & \Gamma ::= & \epsilon \mid \Gamma,\ x :\ T \mid \Gamma,\ t =\ T \mid \Gamma,\ x\ \mapsto\ \overline{S_c}
\end{array}
$$

Figure 4.4: A subset of the grammar for Declarations and top-level definitions, as well as types and typing context.

ROSpec is designed based on the concepts identified in Table 4.1. Of these, we defined the system, node types, and node instances as top-level definitions within a ROSpec file. A **Writer** declares the node types, while an **Integrator** declares the system and its node instances.

Users can include multiple declarations within each node type or instance: mandatory and optional parameters, pub-sub connections, QoS, and name remappings. These concepts are present in many misconfigurations from prior work [21]. The grammar of ROSpec is defined in Figure 4.4.

ROSpec is a typed language where nodes, parameters, topics, and messages have types. The language includes primitive ROS types, such as `int`, `bool`, `float`, and `string` and their bit-width variations (e.g., `int8`, `float64`), omitted in the grammar. It also has user-defined types (`t`), defining custom message types such as `geometry_msgs/Twist`. The type variables are replaced with the concrete type they alias. In the case of message types, like `geometry_msgs/Twist`, the concrete type is a **struct**, containing named and typed fields, similar to C's structs. Support for optional parameters and arguments is provided through the Optional type, which annotates a type with a default value when the caller omits it. This feature resembles Python's use of default arguments.

Liquid types are the main engine used for modeling semantic properties [117], available whenever a regular type can occur. A type `T` can be annotated with a refinement (`x : T where { e }`) where e can refer to x, restricting its possible values. The traditional Liquid Types style draws refinements from a decidable logic. While this limitation restricts what users can model, we show in Section 4.6 that these can be expressive enough to detect several real-world misconfigurations.

To support type-checking, we rely on a type context (Γ) that contains three types of mappings: a) mappings from variables to types ($x : T$), used for node types, plugin types, and instances; b) type alias information ($t = T$) used to save the human-readable name of a given structure type; and c) connection and transform mappings between nodes and topics, such as representing that a given node `publishes to` or `subscribes to` a topic, and `broadcast` or `listens` to a transform.

4.5.2 Formation Rules

The typing rules in Figure 4.5 validate that a given specification is correct. We highlight in the core premises explained in the main body of the text, also highlighted with the corresponding color.

At the top level, we can have node type, node instance, or system definitions. Node types introduce in the context a mapping from the node type name x to its type (`D-NodeType`).

Node instances are validated using the information from the node type already present in the context (`D-NodeInstance`). The node type needs to be previously defined, and the parameters declared in the instance must match with those declared in the node type (subtyping). These checks validate properties `Typing`, `Bounds`, `Dependency`, `Presence`, `Consistency` and `Conditionals`. Additionally, these rules introduce concrete topic connections to the context by instantiating the node type connections ($\overline{S_c'}$) with concrete values (σ) and applying all remappings in the instance ($\overline{S_r}$).

At the system level, we validate rules that concern connections between nodes and topics, such as `Connection`, `Message`, `Fields`, `TF-Listen`, `TF-Graph`, `Cardinality` and `QoS`. In `D-System`, we validate each node instance that in the system, and then collect the resulting context (Γ') containing all connections between nodes and topics. Given that information, we check that topics with at least one subscriber have a publisher with a matching message type, or the expected number of publishers/subscribers; corresponding publishers and subscribers have compatible QoS settings, according to the ROS2 specifica-

D-NodeType

$$\frac{}{\Gamma \vdash \textbf{node type}\ \ x\ \{\ \overline{S_p}; \overline{S_c}; \overline{S_{fr}}\ \} \dashv \Gamma,\ x : \texttt{NodeT}(\overline{S_p}; \overline{S_c}; \overline{S_{fr}})}$$

D-NodeInstance

$$\frac{\Gamma \vdash x2 : \texttt{NodeT}(\overline{S'_p}; \overline{S'_c}; \overline{S_{fr}}) \qquad \Gamma \vdash \overline{S_{p_i}} <: \overline{S'_p} \qquad \sigma = \overline{S_{p_i}} \cup \{x \mapsto e \mid x : \texttt{Optional}(T, e) \in \overline{S'_p},\ x \notin \overline{S_{p_i}}\}}{\Gamma \vdash \textbf{node instance}\ \ x_1 : x_2\ \{\ \overline{S_{p_i}};\ \ \overline{S_r}\ \} \dashv \Gamma,\ x_1 \mapsto\ \overline{S'_c}[\sigma][\overline{S_r}]}$$

D-System

$$\frac{\begin{array}{c}\Gamma \vdash \overline{D} \dashv\ \Gamma' \\ \forall s \mapsto \textbf{subscribes to}(x,\ T_2) \in \Gamma',\ \exists\, p \mapsto \textbf{publishes to}(x,\ T_1) \in \Gamma' \wedge \Gamma' \vdash T_1 <: T_2 \\ \forall s \mapsto \textbf{publishes to}(x)\ \textbf{with}\ qos(q1) \in \Gamma', \\ \forall s' \mapsto \textbf{subscribes to}(x)\ \textbf{with}\ qos(q2) \in \Gamma',\ \texttt{check\_qos}(q1,\ q2) \\ \forall\ s \mapsto \textbf{listens}(x_1, x_2) \in \Gamma',\ \exists\, p \mapsto \textbf{broadcasts}(x_1, x_2) \in \Gamma' \\ \forall\ s \mapsto \textbf{broadcasts}(x_1, x_2) \in \Gamma',\ \forall\ s' \mapsto \textbf{broadcasts}(x_2, x_3) \in \Gamma', x_1 = x_2\end{array}}{\Gamma \vdash \textbf{system}\ \{\ \overline{D}\ \} \dashv\ \Gamma'}$$

Figure 4.5: Definition Formation Rules. $\boxed{\Gamma \vdash D \dashv \Gamma}$

tion;[13] nodes expecting a TF transform from x_1 to x_2 have a node publishing that transform; and there is only one node broadcasting information to a child frame.

S-Int

$$\frac{}{\Gamma \vdash \texttt{int} <: \texttt{int}}$$

S-Bool

$$\frac{}{\Gamma \vdash \texttt{bool} <: \texttt{bool}}$$

S-Struct

$$\frac{\overline{\Gamma \vdash T_i <: U_i}}{\Gamma \vdash \textbf{struct}\{\ \overline{x_i : T_i}\ \} <: \textbf{struct}\{\ \overline{x_i : U_i}\ \}}$$

S-Optional

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash T <: Optional(U, e)}$$

S-Param

$$\frac{\Gamma \vdash self(e) <: T}{\Gamma \vdash \textbf{param}\ x = e <: \textbf{param}\ x : T;}$$

S-Where

$$\frac{\Gamma \vdash T <: U \qquad \Gamma, x_1 : T \vdash e_i \implies e_2[x_2 \mapsto x_1]}{\Gamma \vdash (x_1 : T\ \ \textbf{where}\ \ \{\ e_1\ \}) <: (x_2 : U\ \ \textbf{where}\ \ \{\ e_2\ \})}$$

Figure 4.6: Subtyping Rules. $\boxed{\Gamma \vdash t_1 <: t_2}$

[13] https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html

Figure 4.6 shows the subtyping rules. Rules are standard for basic types, optional types, and structs (all fields must match, and subtyping is covariant). S-Param relies on selfification to convert a concrete value into a refined type that only describes itself [106]. S-Where dispatches implications to either evaluation, when e_1 is made of variable assignments (e.g., $x = v_1 \wedge y = v_2$), or dispatched to a Satisfiability modulo theories (SMT) solver, like z3 [35], (e.g., $x > 1 \implies x > 0$) which only occurs in message types [117].

4.6 Evaluation

To demonstrate ROSpec's ability to specify ROS components and their integration, we evaluate the language in two ways: (1) We model a medium-size robotics case study system to show that the language can model an entire robot when looking at the macro perspective of the systems; and (2) We model the dataset of misconfigurations so that we can understand the coverage of the language, its limitations, and further work needed to address them.

4.6.1 Case Study: Neobotix MP-400 in AWS Small Warehouse

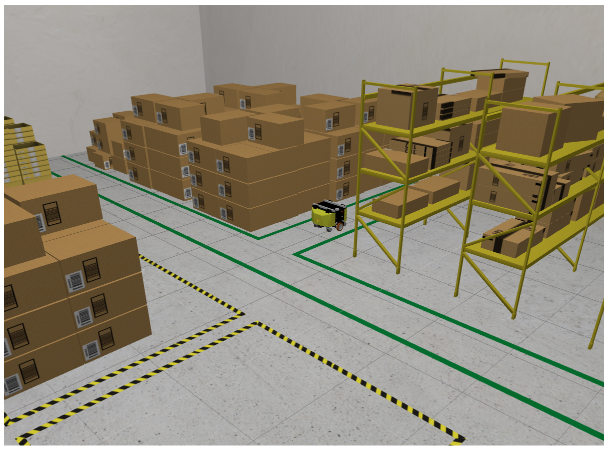


Figure 4.7: AWS Small Warehouse World.

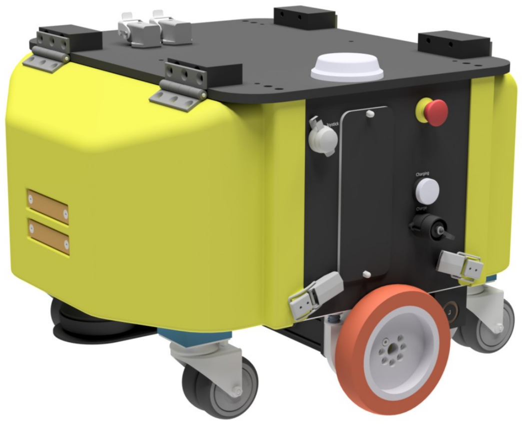


Figure 4.8: Neobotix MP-400.

System Details

Our case study was taken from a popular online robotics learning platform, The Construct Sim,[15], which offers ROS-based courses ranging from basic concepts to complex applications using navigation, manipulation, and control. Given that novices face challenges when learning ROS [20], we expect that providing ROSpec specifications to platforms like The Construct Sim can help newcomers detect and understand misconfigurations while learning core ROS concepts.

[14]https://docs.nav2.org/configuration/packages/configuring-amcl.html
[15]http://theconstruct.ai

```
type alias LaserModelType: Enum[Beam, LikelihoodField,
   LikelihoodFieldProb]

node type amcl_type {
    context distribution: AfterHumbleVersion;

    param robot_model_type: Enum[DifferentialMotionModel,
       OmniMotionModel];
    param scan_topic_name: string;
    param map_topic_name: string;

    optional param z_hit: double = 0.5;
    optional param z_max: double = 0.05;
    optional param z_rand: double = 0.5;
    optional param z_short: double = 0.005;
    optional param always_reset_initial_pose: bool = false;
    optional param laser_model_type: LaserModelType = LikelihoodField;

    @qos{sensor_data}
    publishes to particle_cloud: nav2_msgs/ParticleCloud;

    @qos{sensor_data_profile}
    subscribes to content(scan_topic_name): RestrictedLaserScan;

    @qos{system_default_qos}
    subscribes to initialpose: geometry_msgs/PoseWithCovarianceStamped
                                    where {count(publishers(_)) == 1};

    @qos{transient_reliable_qos}
    subscribes to content(map_topic_name): nav_msgs/OccupancyGrid;

    provides service reinitialize_global_localization: std_srvs/Empty;
    provides service set_initial_pose: nav2_msgs/SetInitialPose;

    broadcast map to odom;
    broadcast odom to base_link;
    broadcast base_link to scan;
} where {
    laser_model_type == Beam -> z_hit + z_max + z_rand + z_short == 1;
    laser_model_type == LikelihoodField -> z_hit + z_rand == 1;
    always_reset_initial_pose -> exists(initial_pose);
}
```

Listing (4.11) Writer partial specification for the AMCL node that uses a particle filter to estimate a robot's pose based on sensor data and a known map.[14] The complete case study code is provided in the artifact.

From The Construct Sim course selection, we chose the "Advanced ROS 2 Navigation" course and its respective case study, as the Navigation stack represents one of the most widely used packages in ROS, responsible for path planning, localization, and obstacle avoidance. The course requires students to configure and integrate commonly used Nav 2 components, including AMCL (Adaptive Monte Carlo Localization), keep-out zones, and speed limit filters.

The course setting resembles a real-world Amazon warehouse environment (Figure 4.7) where robots navigate and transport loads [111]. The Neobotix MP-400 (Figure 4.8) used in the case study is a simplified version with a simpler warehouse environment and robotic system, where its primary objective is to navigate between two points in the warehouse using a waypoint follower, avoiding obstacles and dangerous areas, and respecting speed limit zones. The robot is equipped with a 2D LIDAR scanner that provides range measurements to objects in its environment map, which is provided statically to the system.

Component Specification and System Integration

When specifying components using ROSpec, we drew from two primary information sources: source code and online documentation. The source code provided details regarding connection information and quality of service settings, while documentation provided semantic information about configurations.

When modeling the components, we focused on the ones that required configuration in the course: `amcl` for localization, `controller_server` for trajectory execution, `planner_server` for path planning, `map_server` for environment map management, `costmap` for obstacle representation, and `gazebo` for simulation. Additionally, we modeled 13 different plugins used by nodes, i.e., dynamically loaded components that extend the functionality of nodes, through the use of **`plugin type`**, including FollowPath, KeepOutFilter, and SpeedFilter. The `Writer` specification comprises 434 lines of code across 19 different components, while the `Integrator` system instance requires only 64 lines, as many default values for optional parameters remain unchanged.

Listing 4.11 presents a partial specification of AMCL, a ROS 2 package that uses a particle filter to determine a robot's position using sensor inputs and a given map. In this specification, we define contextual information, parameters, connections, TF frames, and parameter dependencies.

This version of the component contains contextual information regarding its `distribution` (Line 4), as there is one parameter name changed after the `Humble` version — other components must also explicitly declare their distribution and ensure it respects this versioning requirement, preventing mismatching component versions.

The specification also defines a custom **`type alias`** (`LaserModelType`) to restrict the set of allowed values for the `laser_model_type` parameter. In ROS, these options are encoded as strings or integer literals later mapped to the respective values. However, this mapping is prone to typing errors or the use of invalid integers, leading to the use of incorrect parameters and having the system execute with default values without the developer's knowledge.

For connections, the AMCL specification describes the topics it **`publishes to`** and `subscribes to` and the `service` it provides. The specification also restricts the number of publishers to the `initialpose` topic (Line 26). `initialpose` is a topic that sets a robot's starting position for localization and should have a single publisher to prevent conflicts from multiple sources overriding the pose. By restricting the number of publishers using the internal language functions, `count(publishers(_)) == 1`, we ensure that only one component publishes to that topic in the system. This refinement extends liquid types beyond their traditional use, allowing reasoning about the system architecture.

Finally, the AMCL documentation presents parameter dependencies that are not enforced when developers configure their system. In ROSpec, we model these in the `node type where` clause, preventing value dependency mismatches. For instance, depending on the `laser_model_type`, the `z_*` parameters used are different, and their sum must be equal to 1 (Lines 37-38).

In summary, we used all language features when specifying components from the case study using ROSpec. Translating the natural language specifications from documentation and incorporating restrictions and dependencies into parameters was generally straightforward. However, defining TF transforms and publisher–subscriber, service, and action connections was more challenging. These are often undocumented and required the manual inspection of multiple source files to understand their interaction with parameter files. From a `Writer`'s perspective, we expect the specification of the component to be easier, as they are already familiar with it.

4.6.2 Misconfiguration Modeling

Dataset Details

To evaluate the language's ability to address different categories of misconfiguration, we used the dataset comprising 50 different types of misconfigurations identified in prior work [21]. The dataset consists of 182 questions from ROS Answers, each annotated with one or more misconfiguration categories. Currently, this represents the most comprehensive available dataset of documented misconfigurations in ROS-based systems. Although this dataset informed our initial language design methodology, evaluating the language against this dataset allows us to assess the language coverage of the different categories of misconfigurations while highlighting specification challenges, current limitations, and potential future extensions.

Dataset Partial Specification

For each ROS Answers post, we manually analyzed questions and answers to understand the components used, attempted integration, and sources of misconfigurations. We then categorized each question according to the five categories presented and described in Table 4.3. When enough context was provided, and the question was within scope, we partially specified the components used in the question and their system instances. Partial specifications are provided due to the extensive manual effort to specify numerous configuration parameters and connections, incomplete documentation requiring manual analysis of scattered source

Table 4.3: Summary of results from manual analysis and specification of 182 questions. Questions may contain multiple misconfigurations; partial specifications and system integrations are provided where sufficient context exist. Complete dataset with questions and specifications is provided in the artifact.

Category	**Description**	**# of Questions**
Detectable	Component specification and integration possible; thus making the misconfiguration detectable.	61 (33.5%)
Documented	Component specification provided, but lacking information for integration; thus acting as documentation.	23 (12.6%)
Not supported	Component information and integration provided, but the language cannot support the misconfiguration.	39 (21.4%)
Out of scope	Questions not applicable to ROS 2, bugs in components, with no misconfigurations, or documentation-related.	31 (17.0%)
Not enough context	Missing information regarding the component preventing its specification and integration.	28 (15.4%)
Total		182 (100%)

code and configuration files, or missing information preventing the complete specification. A question was considered detectable when all misconfigurations annotated in a question were addressed, while not-supported questions indicated that at least one misconfiguration could not be fulfilled by ROSpec. Questions are considered documented with specifications when verification is not possible due to missing information of non-faulty components and their integration into the system. We provide partial specifications that serve as documentation for the available components. Out of scope questions are annotated in the original dataset as *Documentation*, and *Components* and *Infrastructure* bugs — outside of scope of component configuration and integration. Our final ROS misconfiguration dataset provides a mapping between ROS Answers questions, the annotated misconfigurations, and their corresponding writer and integrator specifications.

Results

Table 4.3 presents the language's overall coverage for each category. The categories are grouped by the ability to provide specifications: the first three represent attempted specifications, while the latter two indicate cases where attempting to provide specifications is not possible.

When considering *Out of scope* questions, the majority of the questions are related to how-to use a component questions (11/31) or to bugs internal to components (11/31). The remaining questions are related to ROS 1 concepts no longer applicable to ROS 2, and calibration runtime questions, where the configurations are correct but required some improvements.

When analyzing *Not supported* questions, there are three major features that ROSpec currently does not support: (1) URDF configuration files, responsible for describing the physical and kinematic structure of a robot; (2) detection of race condition misconfigurations in launch files; and, (3) frequency and synchronization properties, as the language does not support specifying the frequency of connections and their synchronization.

The *Detectable* and *Documented* categories cover 46.1% of the total questions — expected, as ROSpec is designed to cover these misconfigurations. When considering the attempted cases, ROSpec successfully covers 68% (84/123) of the questions. ROSpec covers all twelve high-level categories of misconfigurations from prior work, although some sub-categories (e.g., URDF) are completely not covered and some partially covered (e.g., time-related misconfigurations).

In *Documented* questions, **`type alias`** and **`message alias`** play an important role, as they provide semantic information when documentation is missing or unclear. For instance, developers often question physical units in message fields and their dependencies, which we model by creating different message and type aliases and establishing dependencies between specific image encodings and data physical unit types.[16]

In *Detectable* questions, almost all questions required the definition of a **`node type`** (60/61), where most of these contained information about their parameters (38/61), specifications on the dependency between them (27/61), followed by TF transforms (15/61). From the language features proposed, two are the least used to specify components: (1) Quality of Service settings (2/61), as most questions in the dataset relate to ROS 1 where the concept was not introduced, and (2) contextual information (11/61), since questions may be related to contextual information but the specification does not require the **`context`** concept to detect it. Regarding connections, publisher-subscriber is the most used concept with a total of 59 connections (23/61), validating the results from prior work [123], followed by services (4/61) and no actions.

Considering the prevalence of type features in Documented and Detectable questions, liquid and dependent types appear in 40 of the 84 questions (47.6%). Dependent types are more frequently used for defining dependencies between parameters and contextual information, appearing in 27 questions (32.1%), while liquid types are explicitly used in 14 questions (16.7%). This analysis excludes type aliases used with refinement types and inherent refinements in receiver connections, such as subscribers that implicitly expect one publisher.

In summary, the manual analysis of the dataset provided insights on ROSpec specification abilities and limitations. As in the case study, specifying parameters and argument configurations was straightforward, whereas contextual information required more understanding of the application and execution environment. Furthermore, as questions often described the **`publishes to`** and **`subscribes to`** connections, it helped specify components integration. For Documented, constructs like **`type alias`** and **`message alias`** helped provide semantic meaning, such as physical units, for basic types. However, the language still

[16] https://answers.ros.org/question/209450, and https://answers.ros.org/question/260640.

lacks expressiveness specifying some ROS concepts: a) launch file race conditions, which require reasoning about execution order; b) frequency and synchronization, not yet modeled but planned for future extensions; and, c) URDF files, whose extensive configuration files are related to the physical system and not the software perspective.

4.7 Discussion & Future Work

Detecting configuration errors is critical in robotics, as they can lead to erratic and potentially dangerous system behaviors. With ROSpec, we provide developers a means to specify and ensure correct component configuration and integration. In this section, we discuss current limitations, potential language extensions for detecting misconfigurations, and future work to verify source code correctness and improve the developer experience when creating and maintaining specifications.

Language limitations regarding unsupported misconfigurations The evaluation of ROSpec on questions about misconfigurations raised three main limitations in the language.

First, URDF configurations are not supported in the language as these may contain hundreds of physical-related parameters. We considered that specifying these alongside supported elements would hide them as they would represent a small fraction of the specification compared to URDF parameters. A possible solution is extending the language specifically to URDF, and allow developers to provide these separately from the regular component and system configurations.

Second, the language does not support frequency misconfigurations. Questions often did not provide information regarding component frequencies, or when existing, the message processing frequency by components is impacted by hardware limitations, making it challenging to detect.

Third, launch files race condition misconfigurations are not detectable by ROSpec. The language provides a *static* view of the system when all components are already integrated and executed. However, in ROS, this architecture may differ depending on the ordering of launching components. For instance, a consumer service may be launched before a provider service. Detecting such instances requires temporal logic [116] to describe the launch ordering of components.

Liquid and dependent types for configuration and architectural verification ROSpec demonstrates how liquid and dependent types can be used beyond code verification for the purpose of specifying component configurations and their integration. While dependent types allowed the specification of dependencies between configurations and contextual information, liquid types provided restrictions over components configuration values and connections when integrated into a system. Our approach opens opportunities for using these established programming language concepts as verification engines in other architectural domains where configuration correctness is critical. For machine learning pipelines, architectural-level refinements can restrict component ports, preventing classifiers from training on imbalanced datasets or applying temporal aggregation to non-time-series data [34]. Microservices architectures could benefit from extending JSON-based descriptions like the Microservices Architecture Language (MIRL) [86] with liquid types to refine service nodes and their relationships. Similarly, IoT and edge computing architectures [62] could use liquid and dependent types for formal specifications of resource constraints (e.g., memory, bandwidth, and latency) and security requirements [79, 82]. However, further research is needed to validate the effectiveness of the language paradigms within these domains.

Language extensibility to new features As ROS is a real evolving ecosystem, it is critical for ROSpec to adapt and support new concepts and the detection of new misconfigurations. While ROSpec focuses on core ROS concepts, the language is extensible to new features and verifications by reusing contextual information, creating new policies, or implementing new uninterpreted functions. For example, developers can specify deployment-specific information and RMW implementations using the **`context`** keyword, allowing the reasoning of resources and networks. Additionally, messages' timeliness requirements can be implemented through a `frequency` uninterpreted function, refining expectations and guarantees regarding message frequency. Finally, security requirements[17] can be modeled by creating new policies attached to components, ensuring that only trusted components perform authentication-specific operations.

Usability argument for ROS-based domain-specific languages When designing ROSpec, we studied and used domain knowledge and misconfiguration sources to improve language usability.

In ROS, we identify two primary stakeholders in its ecosystem: **`Writer`** and **`Integrator`**. ROSpec explicitly models these perspectives and their respective concerns. This stakeholder specialization separates concerns, allowing them to focus on their role during specification.

Familiarity with the domain concepts can make system description more intuitive for developers. Prior work in domain-specific architectural languages motivates the need for domain-specific concepts to improve language usability [147]. When designing ROSpec, we incorporated ROS 2 concepts to provide familiarity with concepts developers use, while drawing insights on prior research [21] to ensure language expressiveness to detect misconfigurations.

When designing a language, ergonomics are important to reduce the friction between the language and the domain concepts [136]. In designing ROSpec, we minimized this friction by: (1) identifying distinct stakeholder roles and incorporating their perspectives; (2) building specifications upon established ROS concepts familiar to developers, reducing the learning curve; and (3) adopting ROS naming conventions for consistency (e.g., a **`node publishes to topic`**).

Nevertheless, effectively evaluating a language's usability requires interviews with ROS developers, as they may share the same usability challenges when using liquid types as identified in prior work [51]. These provide the understanding of how they specify their components, identify their challenges, and understand the features needed for specification and misconfiguration detection.

[17]https://design.ros2.org/articles/ros2_dds_security.html

Leveraging architectural recovery to ensure source code to specification consistency ROSpec is a language that abstracts ROS implementation details and provides specifications for component configurations and their integration. This verification ensures correct configuration and interaction between components (i.e., external specification consistency). However, ROSpec does not verify the correctness of source code against specifications (i.e., internal consistency).

Checking internal consistency in ROS is challenging as component configurations are distributed across different file formats and programming languages (e.g., Python and C/C++). Nevertheless, recent improvements in architectural recovery analysis tools such as HAROS [125], ROSDiscover [133], and ROSInfer [40] can collect configurations and identify component connections. Recent advances in misconfiguration detection — including test-based approaches [26], static and dynamic analysis [64, 65, 155], and Large Language Models [87] — could complement these architectural tools to identify misconfigurations at the source-code level.

However, as discussed in Section 4.2, these tools often lack developer intent, limiting their ability to perform meaningful verifications. By leveraging these tools, we can verify components' internal consistency by inferring their configurations. Any mismatches indicate that the component does not respect the specification, either because the implementation is out of sync or due to a bug.

Automated synthesis of specifications from source code and natural language Writing specifications can be repetitive and challenging, as developers must duplicate their configuration efforts in the specification while translating their natural language requirements into formal language properties. We can automate portions of component specification writing and generate initial specification templates by leveraging architectural recovery static analysis tools alongside natural language processing approaches, particularly Large Language Models (LLMs). Architectural analysis tools have statically recovered configurations and connections from source code [40, 125, 133]. Concurrently, LLM-based techniques have shown promise in converting natural language specifications into formal specifications [59, 141]. Combining these approaches can reduce specification writing effort. Moreover, when specifications and implementations diverge, we can automatically update specifications by computing differences between implementation versions and their specifications.

Runtime verification of non-statically verifiable properties ROSpec allows developers to introduce specifications that verify correct component configuration and integration. However, certain specifications, such as those defined in **`message alias`**, cannot be verified internally against the component implementation. These specifications require runtime execution to verify whether transmitted and received data dynamically conform to the constraints, mainly when components interact with sensors that collect real-world data. By leveraging component specifications as expressions of developer intent, we can automatically generate runtime monitors [17, 46, 66] that detect misconfigurations against properties defined in the message field types.

4.8 Conclusion

Misconfigurations in ROS-based systems can lead to unpredictable and potentially dangerous system behaviors. In this paper, we introduced ROSpec, a domain-specific language for specifying component configurations and their integration. We grounded our approach in a study of domain concepts and misconfiguration properties, allowing developers to specify their components and ensure their correct integration. Our evaluation demonstrated the language's abilities by modeling a warehouse robot and addressing misconfigurations identified in prior work. While ROSpec represents a step towards more reliable robot software, future work will focus on evaluating the language usability while improving the language features. By addressing these challenges, we aim to improve the correctness and safety of complex robotic systems.

Chapter 5

Architectural Evolution and Drift Analysis in Open Source Robotics (Proposed Work)

In the previous chapter, I presented a domain-specific language to detect real-world misconfigurations in ROS-based robot software. The proposed approach, along with prior work in architectural description languages [10, 11] and domain-specific languages [121], specify component configurations and architectures in ROS systems. While these approaches help developers detect specific categories of misconfigurations, they rely on the assumption that ROS configurations and architectures remain static throughout the system's lifetime.

In practice, robot software evolves. Developers add new components, modify configuration, and change communication settings to meet changing requirements and introduce new features. This evolution forces developers to do the burdensome task of manually updating specifications to keep them aligned with the source code. When specifications fall out of sync with the implementation, the system experiences architectural drift.

Architectural Drift occurs when changes to a system are made that do not reflect the documented architecture, leading to inconsistencies between implementation and documentation [108]. For instance, In ROS-based systems, when documentation becomes stale, developers rely on outdated or incorrect assumptions about how components should be configured and integrated, leading to misconfigurations [21]. Given the introduction of new specification languages and that robot software must co-evolve with their documentation to ensure correct use, techniques are needed to automatically detect and prevent architectural drift.

However, no prior work has systematically studied how components and architectures evolve in highly-configurable robotic systems like ROS, nor how documentation keeps pace with this evolution. Understanding this co-evolution is critical for two main reasons. First, it reveals which architectural constructs undergo the most frequent changes, providing insights into the design of source-code analysis tools. Second, it measures the documentation maintenance burden that developers face as systems evolve, to understand if specification are sustainable in practice as systems grow in complexity.

To address this gap, I propose to conduct a large-scale empirical study of architectural

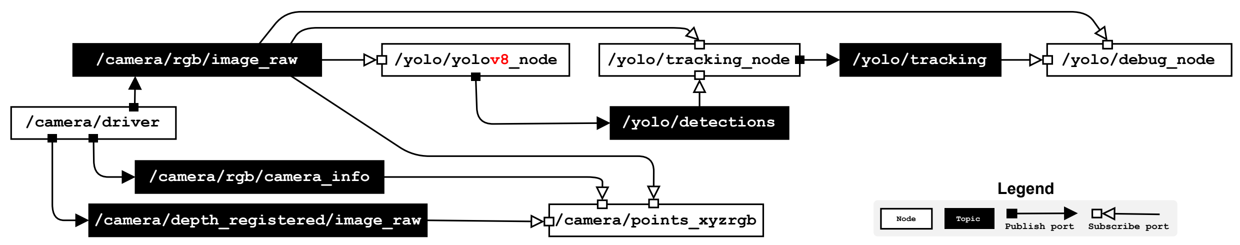


Figure 5.1: Architectural view of the publisher-subscriber graph documented in the mgonzs13/yolo_ros project. The package was refactored and references to *v8* were removed. However, current documentation does not reflect the implementation.

evolution and drift in open-source ROS systems. I systematically analyze how ROS-based robot software architectures evolve over time across multiple releases and quantify the extent to which implementations drift from their documentation. This approach consists of four main steps. First, I will collect a dataset of open-source ROS projects with tagged releases, which provide stable, well-defined snapshots of system evolution. Second, I will develop a lightweight architectural recovery tool that extracts configuration and architectural constructs from both ROS 1 and ROS 2 systems, supporting both C++ and Python implementations. Third, I will track how documentation evolves alongside implementation by analyzing README files, images, and wikis across publicly available versions. Finally, I will apply quantitative analysis to measure evolution patterns (e.g., frequency of changes, types of modifications) and qualitative analysis to understand the nature of changes, and causes of architectural drift. By understanding how ROS-based systems evolve and documentation drifts, I can provide insights into developing languages and automated tools that account for the evolving-nature of these systems.

Through this work, I propose to make the following contributions:

1. **An empirical study of architectural evolution in ROS systems.** I analyze how frequently architectures change, what types of changes happen (e.g., adding components, modifying parameters, restructuring communication), and which architectural elements change the most across releases in open-source ROS projects.
2. **An analysis of architectural drift in ROS systems.** I measure the extent to which documentation accompanies implementation changes, identify which types of architectural elements drift most frequently, and the reasoning underlying the drift.
3. **A cross-version architectural drift analysis tool for ROS.** I develop an automated tool that generates a report for developers to automatically detect architectural drift by extracting architectural elements from both ROS 1 and ROS 2 systems in C++ and Python, and comparing these against documentation.

5.1 Methodology

To understand architectural evolution and drift in ROS systems, I propose to conduct a large-scale empirical study of open-source ROS projects. This section presents the research questions, describes the proposed data collection and analysis approach (Section 5.1.1), and discusses threats to validity (Section 5.1.2).

To understand architectural evolution and drift in ROS systems, I pose two main research questions.

RQ1: To what extent do configurations and architectures evolve across releases in ROS systems?

Before I can detect or prevent architectural drift, one must first understand how frequently and in what ways do ROS architectures actually change. Prior work on ROS misconfigurations has focused on single snapshots of systems [40, 133], providing no insight into if architectures remain stable or the extent the maintenance of different architectural elements may impact the performance of software analysis tools. Without quantifying the rate and nature of architectural changes, I cannot assess if manual documentation maintenance is sustainable, nor can I design automated source-code analysis detection tools that target common evolution patterns. Furthermore, understanding architectural evolution also reveals if certain types of projects experience more rapid change, helping prioritize where automated tooling is most needed.

To systematically study these dynamics, I propose the following sub-research questions.

RQ1.1: How does evolution differ between ROS 1 and ROS 2? ROS 2 introduced architectural changes, including a new middleware layer (DDS) and different communication patterns. These differences may lead to distinct evolution patterns, as developers adapting to ROS 2's new paradigms might restructure their architectures differently than when working within ROS 1's conventions. Understanding platform-specific evolution patterns is critical as the community continues migrating from ROS 1 to ROS 2.

RQ1.2: How does project age affect evolution patterns? I compare evolution in older, mature projects versus newer projects to understand whether architectural churn decreases as systems stabilize. Mature projects may exhibit fewer but more carefully considered changes, while newer projects might undergo rapid architectural experimentation. This insight informs when in a project's lifecycle drift detection tools provide the most value.

RQ1.3: How does evolution differ between official distribution packages and community projects? Projects in official ROS distributions may face different maintenance pressures than independent GitHub projects. Distribution packages must maintain backward compatibility and adhere to community standards, potentially constraining their evolution compared to independently developed community projects that have more flexibility. Understanding these differences helps tailor documentation practices and tooling to different development contexts.

RQ1.4: How do project characteristics correlate with architectural changes? I examine whether factors such as system size, maturity, team structure, and popularity influence the frequency and types of architectural changes across releases. Understanding these correlations helps identify which types of projects are most likely to experience rapid architectural evolution and thus face greater documentation maintenance challenges.

RQ2: To what extent does documentation accompany configuration and architectural changes?

Prior work on ROS misconfigurations [21] found that outdated documentation leads developers to make incorrect assumptions about component usage, resulting in dangerous misconfigurations. However, the scale and patterns of documentation drift remain unmeasured. Without empirical evidence of which architectural elements drift most frequently, when drift occurs relative to implementation changes, and which types of projects experience the most severe drift, researchers cannot prioritize which aspects of the problem to address first, and practitioners lack benchmarks to assess whether their documentation practices are adequate.

I propose to study documentation drift through the following sub-research questions.

RQ2.1: Which architectural elements are most prone to drift? I measure drift rates for different architectural constructs—topics, parameters, TF frames, nodes, and embedded DSL configurations—to identify which elements most frequently become stale. Certain architectural elements may be more difficult to document, change more frequently, or lack adequate tooling support, leading to higher drift rates. Identifying these high-drift elements allows automated detection tools to prioritize where they can provide the most value.

RQ2.2: How do drift patterns vary across project types? Similar to RQ1, I compare drift rates between ROS 1 and ROS 2, older and newer projects, and distribution packages versus community projects. This comparison reveals whether contextual factors such as platform maturity, project age, or maintenance standards affect how well documentation keeps pace with implementation changes. Understanding these patterns helps target interventions to the contexts where drift is most problematic.

RQ2.3: What is the lag between implementation changes and documentation updates? I quantify how quickly documentation is updated after architectural changes, if at all, to understand the temporal dimension of drift. Some projects may update documentation immediately, while others may accumulate changes across multiple releases before updating documentation, or never update it at all. Measuring this lag reveals whether drift is primarily a problem of delayed updates or complete neglect, informing different mitigation strategies.

5.1.1 Data Collection and Analysis

Figure 4.2 presents an overview of the proposed approach, which consists of five main stages: dataset collection, architectural recovery, documentation collection, drift analysis, and quantitative/qualitative analysis.

Dataset Collection. I begin with the ROS Anatomy dataset [8], which contains over 100,000 ROS open-source repositories from GitHub spanning both ROS 1 and ROS 2 systems in C++ and Python. ROS Anatomy provides metadata about each repository, including its primary ROS version, number of releases, and primarily used programming language. To focus on projects with stable evolution, I filter for repositories containing at least 10 tagged releases. Releases represent stable snapshots where developers consider the system ready for use, providing well-defined points for measuring evolution. I empirically select projects with fewer than 10 releases as these tend to exhibit high "*entropy*" in early development stages as developers experiment with initial designs, making it challenging to distinguish architectural evolution from initial prototyping. This filtering yields 287 repositories for analysis.

Architectural Recovery. For each repository, I use the proposed architectural recovery tool (Section 5.2) to extract configuration and architectural elements from the source code at each release. Although, prior work on architectural recovery provides a more sound and complete approach [40, 133], it only supports ROS 1 C++ versions. The proposed approach considers a cross-language and ROS version analysis with the tradeoff of its precision. By recovering architectures across all releases for each project, I obtain a timeline that reveals how architectures evolve over time.

To evaluate the accuracy of the configuration and architectural elements recovery tool, I propose to use Cochran formula for small populations with random sampling of the total number of commits from the dataset of 287 repositories.

Documentation Collection. Documentation in ROS systems exists in two primary forms. *In-repository documentation.* Projects include documentation directly in their repositories through README files, images, and other documentation files. This documentation is version-controlled alongside the source code, allowing us to track changes automatically by checking out each release tag and extracting the documentation content. *External documentation.* Some projects maintain documentation on external platforms (e.g., ROS Wiki, external repositories, and project websites). Tracking changes in external documentation is more challenging, as version history may not be preserved or may not align with source code releases. To address this, I manually inspect each repository for links to external documentation and attempt to reconstruct documentation history where possible.

5.1.2 Threats to Validity

External Validity. Our findings may not generalize to all ROS systems. By focusing on open-source projects with at least 10 releases, I may only consider well-maintained and mature projects while not considering early-stage or closed-source systems. However, mature projects are those where architectural evolution and drift are most likely to be problematic, making them appropriate subjects for our study.

Internal Validity. Our architectural recovery tool may produce false positives (detecting architectural elements that do not exist) or false negatives (missing elements that do exist). To mitigate this threat, I designed our tool to recognize both current and deprecated API calls across ROS 1 and ROS 2. However, I acknowledge that our tool cannot detect architectural elements defined through indirect API wrappers or custom abstractions. The analysis tool is modular so any future work in architectural recovery of ROS systems can be plugged and improve the architectural drift analysis.

5.2 Proposed Approach

To measure architectural drift, I must first recover both the implemented architecture from source code and the documented architecture from documentation files at each release.

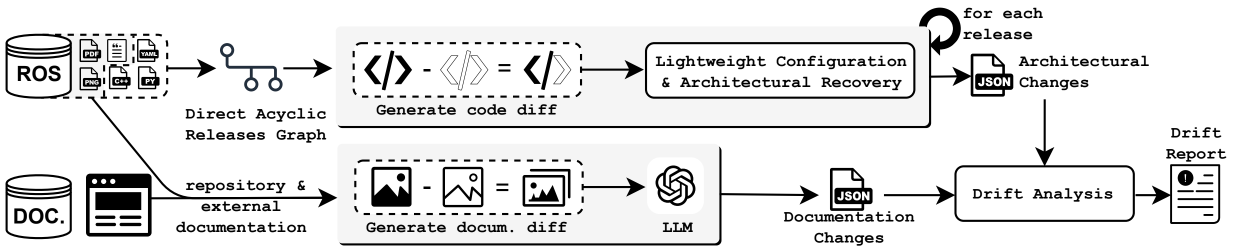


Figure 5.2: Overview of our approach for detecting architectural drift in ROS systems. Given a repository, I construct a directed acyclic graph of releases to identify evolution paths. For each file change between releases, I apply lightweight architectural recovery to source code changes and use an LLM to extract architectural information from documentation changes. Finally, I compare recovered architectures against documented architectures to identify drift and generate a report across all releases.

Figure 5.2 illustrates our approach, which consists of three main components: lightweight architectural recovery from source code (Section 5.2.1), documentation architectural inference (Section 5.2.2), and drift analysis (Section 5.2.3).

5.2.1 Architectural Recovery from Source Code

Constructing the Release Evolution Graph. Given a repository, I begin by constructing a directed acyclic graph (DAG) representing the relationships between releases. Most repositories follow a linear release sequence, but some maintain multiple parallel release branches (e.g., for different ROS distributions or stable versus development versions). By constructing the full DAG, I can analyze all evolution paths rather than assuming a single linear history. For each edge in the graph representing the transition from one release to the next, I identify all files that changed between the two releases.

Extracting Architectural Constructs via Tree-Sitter. ROS is currently transitioning from ROS 1 to ROS 2, with each platform supporting multiple distributions (e.g., ROS 1 Noetic, ROS 2 Foxy, Humble, Iron) that introduce API changes and deprecations. Additionally, the majority of ROS projects are written in either Python or C++ [122], creating four distinct language-platform combinations to support. Prior work on architectural recovery, such as ROSDiscover [133], focuses exclusively on ROS 1 systems written in C++, limiting its application to the entire ROS ecosystem.

To support this diversity while maintaining efficiency, I use Tree-sitter,[1] a parser generator and incremental parsing library that has been successfully applied to code transformation tasks across multiple languages [74]. Tree-sitter provides fast, incremental parsing and a unified query language that works across different programming languages, allowing us to define architectural recovery rules once per construct.

For each changed source file, I apply configuration and architecture-specific queries to

[1]https://tree-sitter.github.io/tree-sitter/

```python
class YoloNode(LifecycleNode):

    def __init__(self):
        super().__init__("yolo_node")

        self.declare_parameter("model_type", "YOLO")
        self.declare_parameter("threshold", 0.5)
        self.declare_parameter("max_det", 300)
        self.declare_parameter("enable", True)

    def on_configure(self, state):
        self.image_qos_profile = QoSProfile(
            reliability = self.reliability,
            history = QoSHistoryPolicy.KEEP_LAST,
            durability = QoSDurabilityPolicy.VOLATILE,
            depth = 1,
        )
        self._pub = self.create_lifecycle_publisher(
            DetectionArray, "detections", 10)

    def on_activate(self, state):
        self._enable_srv = self.create_service(
            SetBool, "enable", self.enable_cb)
        self._sub = self.create_subscription(
            Image, "image_raw", self.image_cb)
```

Listing 5.1: Excerpt of source-code from mgonzs13/yolo_ros for yolo_node. The highlighted ROS API calls create a component, its configuration and connections.

extract ROS API calls. Listing 5.1 presents an example of a node implemented in Python with the respective API calls, while Listing 5.2 shows an example query for detecting ROS 2 node definitions in Python. Since ROS 2 nodes are defined via superclass invocation, I capture the superclass name and verify it matches a known node type (e.g., `Node` or `LifecycleNode`). Each query defines three types of captures for post-processing:

- **`@api`**: The name of the ROS API being invoked (e.g., `create_publisher`, `declare_parameter`);
- **`@argument`**: The architecturally relevant arguments to the API call (e.g., topic names, message types, parameter names and default values);
- **`@api_call`**: The complete API call including all context.

The queries are organized into a taxonomy of architectural constructs derived from prior study of ROS misconfigurations (Chapter 3). Each query is labeled with a high-level category and subcategory, allowing us to automatically classify architectural changes.

- **Components**: API calls related to creating node instances: node, lifecycle-node, composition;

```
(class_definition
  name: (identifier)
  superclasses: (argument_list
    (identifier) @class (#eq? @class "LifecycleNode"))

  body: (block
    (function_definition
      name: (identifier) @fun (#eq? @fun "__init__")
      body: (block (expression_statement (call
    function: (attribute attribute:
          (identifier) @api (#eq? @api "__init__"))
        arguments: (argument_list .(_) @argument .)
      )) @api_call)
    )
))
```

Listing 5.2: Tree-sitter query to capture a lifecycle node initialization in Python for ROS 2.

- **Configuration** : API calls for declaring or accessing configuration values: `parameter` , `argument` ;
- **Transforms** : API calls for managing TF coordinate frame transformations: `listener` , `broadcaster` ;
- **Connections** : API calls defining inter-component communication: `publisher` , `subscriber` , `service-client` , `service-server` , `action-client` , `action-server` .

To compare the recovered architecture with documentation, I normalize captured API calls into a standardized JSON format. For example, component captures are normalized to include a `name` field containing the node name, while parameter captures include both `name` and `default_value` fields. This normalization abstracts away language-specific syntax differences, allowing us to compare architectures across ROS versions and programming languages.

Furthermore, for each release, I also save metadata including commit information (hash, date, authors, parent commits), a complete architectural snapshot listing all constructs present in the release, and a list of architectural changes relative to the parent release(s). This information will allow us to analyze the proportion of changes per release.

5.2.2 Documentation Architectural Inference

Unlike source code, with a well-defined syntax, documentation exists in diverse formats and uses natural language to describe architectural concepts. I address the challenge of infering documentation architecture by distinguishing between internal documentation and external documentation, applying appropriate extraction strategies to each.

Internal Documentation. Internal documentation includes README files, markdown guides, figures, and project wikis stored directly in the repository. Because this documentation is version-controlled, I can track its evolution alongside source code changes by checking out each release tag and extracting documentation content.

For each pair of releases, I identify all documentation files that changed and compute a diff. For PDF documentation, I first extract text and embedded images, then provide the extracted content to the LLM, while for images I provided the model both the previous and after images. I then use a Large Language Model (LLM) to extract architectural information from these changes. Our prompt uses Chain-of-Thought reasoning and few-shot examples to guide the model in identifying architectural constructs mentioned in the documentation. I instruct the LLM to output structured JSON matching the same schema used for source code recovery. This unified format allows direct comparison between implemented and documented architectures. For example, if documentation mentions *"publishes to /cmd_vel topic with geometr_msgs/Twist messages,"* the LLM outputs:

```json
{
  "category": "connections",
  "subcategory": "publisher",
  "name": "/cmd_vel",
  "message_type": "geometry_msgs/Twist"
}
```

External Documentation ROS projects may also maintain documentation on external platforms such as ROS Wiki, GitHub Pages, or Read the Docs. Tracking external documentation presents challenges because version history may not be available or may not align with source code releases.

For externally hosted repositories, I follow the same process as internal documentation, analyzing version history to track changes. For web-based documentation, I use the Wayback Machine API[2] to retrieve historical snapshots of documentation websites. Given a documentation URL, I query the Wayback Machine for all available snapshots and identify the snapshot closest to each source code release date. For each snapshot, I crawl all subpages and extract HTML content. I compute diffs between consecutive snapshots and provide these to the LLM following the same process as internal documentation.

5.2.3 Drift Analysis

Given the summaries of source code and documentation at each release, I systematically detect architectural drift. I define drift as any inconsistency between the implemented architecture recovered from source code and the documented architecture extracted from documentation. I identify two primary types of architectural drift. **Missing documentation drift** occurs when an architectural element exists in the implementation but is not mentioned in documentation. For example, if source code defines a parameter `max_velocity` with default value 1.0, but documentation contains no mention of this parameter, I record missing documentation drift. **Incorrect documentation drift** occurs when both implementation and documentation describe the same architectural element, but with mismatched

[2]https://web.archive.org

details. For example, if source code publishes to `/cmd_vel` with message type `Twist`, but documentation describes it uses `TwistStamped`, I record incorrect documentation drift.

Drift Metric For each release, I compute two metrics to quantify the severity of drift. *Drift rate* measures the percentage of implemented architectural elements with any form of drift (missing, stale, or incorrect documentation). Formally, for a release r with implemented elements I_r and documented elements D_r:

$$\text{Drift Rate}_r = \frac{|\{e \in I_r : e \notin D_r \text{ or } e \text{ mismatches } D_r\}|}{|I_r|}$$

Drift lag measures the temporal gap between when an implementation change occurs and when documentation is updated (if ever). For each architectural change in release r, I search forward through subsequent releases to determine when documentation is updated.

5.3 Conclusion

In this chapter, I propose a large-scale empirical study of architectural evolution and drift in ROS systems. By analyzing how configurations and architectures change across releases and measuring how documentation keeps pace with these changes, we can understand whether maintaining specifications alongside evolving implementations is sustainable in practice. The findings from this study will provide insights for architectural change frequency and documentation drift across the ROS ecosystem. These baselines provide the foundation to develop automated drift detection tools that warn developers when documentation falls out of sync with implementation. By revealing which architectural elements change most frequently and which are most prone to drift, this work also guides both the design of analysis tools and the prioritization of documentation maintenance efforts in practice.

Chapter 6

Specifying ROS-bots with ROSpec: A Usability Study (Proposed Work)

ROS-based robot software is challenging to configure correctly. As seen in Chapter 3, developers introduce misconfigurations when integrating reusable components, leading to unintended and potentially dangerous behavior. These misconfigurations arise from implicit assumptions about their configuration that are neither checked nor documented. To address this problem, I introduced a specification language (Chapter 4) to make these assumptions explicit and allow the detection of misconfiguration prior to deployment.

However, the successful adoption and use of specification languages depends on their usability by practitioners. For instance, languages must be expressive to capture system architectures [69], provide domain specialization to simplify descriptions [147], have an intuitive syntax [97], and abstraction levels that match developer understanding [147].

The robotics domain presents unique usability challenges that distinguishes it from other software engineering domains. Robot software involve distributed communication, real-time constraints, and cyber-physical interactions with the physical world, requiring specifications to capture both software and contextual properties. Moreover, from mechanical engineering, to electrical engineering, and computer science, robot software developers come from diverse backgrounds [1], providing different levels of expertise and usability requirements.

Despite the introduction and use of specification and architectural languages for robot software like ROS [11, 24, 125], no prior work has systematically studied the challenges and usability requirements underlying component-based robot software. Without this empirical study, language designers rely on intuition rather than evidence when designing language's syntax, error messages, and tooling support. **Understanding developer usability challenges is critical for designing languages that are used in practice.**

To address this gap, I propose to conduct a usability study to understand the challenges on the specification of component-based robot software components and systems. To study these challenges, I use ROS, the most popular open-source framework for robotics, as a proxy for the general-purpose component-based robot software development. Moreover, I use the ROSpec specification language, as the most recent specification language for ROS, whose design considers different stakeholders and addresses the misconfigurations identified in Chapter 3.

The proposed study encompasses a task-based observation with semi-structured interviews for the different stages of the development and reasoning of specifications. I propose to recruit ROS developers spanning two main roles: (a) *component writers* who create and maintain reusable ROS components, and (b) *component integrators* who integrate existing components into complete systems. For both stakeholders, I qualitatively and quantitatively measure their ability to write, update and understand ROS component and system specifications.

The expected contributions of this study are described as follows.

1. **An empirical study of specification language usability in ROS-based robot software.** I identify the key challenges developers face when writing, reading, and using specifications for ROS components and systems.
2. **Insights into developer priorities and practices.** I reveal what types of properties developers consider most important to specify, what specification tasks they find most valuable versus burdensome, and how specification may fit into their current development workflows.
3. **Language design requirements for specification languages in robotics.** Based on the study findings, I outline the requirements for improving the usability of specification languages for component-based robot software, covering syntax design, tool support, error messaging, and integration with existing development practices.

6.1 Study Design

Understanding the usability requirements of developers within a specific domain, is essential for the successful adoption of domain-specific languages. Each of the following research questions captures the perspective of componet writers and integrators at different stages of interacting with specifications.

RQ1: To what extent can component writers and integrators write specifications for components? The early stage of specification development encompasses the creation of the initial component and system specifications. The objective of this research question is to evaluate the ease of component and system specification given existing component source code and configuration files.

For component writers, as writing and updating component specifications requires deep understanding on configuration semantics, I request participants to specify one publicly available component they have authored. Understanding the challenges of specification creation is critical for assessing, (1) what specification features participants use the most; (2) which features they struggle with, possibly due to usability barriers; and (3) what information they cannot express, revealing gaps in language expressiveness.

For component integrators, they are required to integrate existing component specifications for the case study system presented in Section 4.6. Given source code and component specifications for the Navigation Stack 2, I request participants to identify the used components, determine how components are connected (e.g., topics, services, remappings), and instantiate the configurations for one component. This task allows us to understand if language's abstraction level is appropriate for describing systems.

For both categories of participants, I propose to measure the participants performance by analyzing specification accuracy by comparing it against the ground truth, their completion time, perceived task complexity, features participants expected but could not use, and the proportion of time spent on different activities (e.g., reading source code, writing specifications, resolving errors).

RQ2: How effectively can component writers convert existing natural language descriptions into specifications? Currently, ROS components may have natural language documentation but no formal specification. In the transition to formal specifications, participants must translate prose descriptions into formal specifications, requiring them to interpret potentially ambiguous documentation and identify relevant specification features. In this research question, I evaluate participants ability in converting existing documentation, identifying which types of changes are most challenging to reflect in specifications. Participants will be tasked with converting a popular component's documentation (e.g., `ros2_control` or `robot_localization`) while I measure translation accuracy (i.e., correct, partial, or incorrect) compared to ground truth, completion time, and perceived difficulty of this translation task.

RQ3: To what extent do component integrators understand components specifications and system architectures? Component and system specifications provide value if component integrators can effectively reason about them. I propose to evaluate specification comprehension through two main tasks to evaluate if specifications support component-level and system-level understanding. First, participants receive component specifications previously created and validated by component authors, and answer questions testing their understanding on component connections, its configurations and restrictions. Then, they examine a complete system specification for a case study system (Section 4.6) and explain the system's architecture and instantiated components configurations. For both tasks, I measure the comprehension accuracy, time to find information, and challenges in understanding the architecture.

6.1.1 Methodology

Participant Recruitment. I propose to recruit ROS developers from varying backgrounds and expertise that reflect the two roles in the ROS ecosystem: component writers and component integrators. To recruit participants for the study, I intend to use two primary approaches. First, ROS Discourse & Social Media where I post recruitment announcements on ROS Discourse (the official ROS community forum) and Robotics Stack Exchange. These platforms reach active community members likely to have specification needs. Second, GitHub mining where I identify repositories with recent commits (within the last year) to popular ROS packages. I extract contributor emails from commit metadata and send recruitment emails highlighting their package. I screen respondents for relevant experience

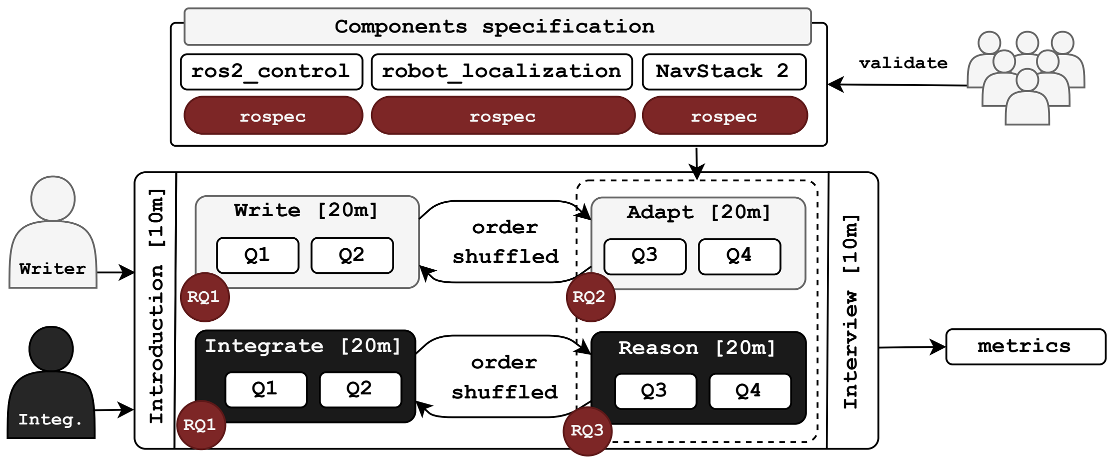


Figure 6.1: Overview of the usability study methodology. Each participant is split between component writer and integrator, according to their experience in writing components or systems. I then perform one-hour sessions with both component writers and integrators, where writers attempt to create and adapt specifications while integrators interpret and reason about existing specifications to guide integration tasks. Finally, I collect qualitative insights through observation and semi-structured interviews, complemented by quantitative measures of task completion and specification correctness.

(minimum 1 year ROS development, specific role as writer or integrator). All participants receive a $30 Amazon gift card for compensation.

Study Protocol. Figure 6.1 illustrates our study design. First, I provide an introduction to the study and the ROSpec specification language. Then, participants are split into one of two conditions according to their experience in creating and maintaining ROS components. Component writers perform specification tasks requiring them to write and adapt specifications for different components. Component integrators are tasked to interpret existing specifications and use them to guide integration tasks for both individual components and complete systems. Throughout these tasks, I observe participants' approaches, record their questions, challenges, and how they use (or avoid using) different specification features. Following the task sessions, I conduct semi-structured interviews to collect feedback on their challenges and recommendations for future improvements.

6.1.2 Threats to Validity

Internal Validity. I identify three main threats to internal validity: task scope, sample size, and language-specific findings. The first threat concerns the use of small components and systems to fit within session time constraints. Specification effort may not scale and usability issues manageable for small components may become challenging in larger systems with hundreds of configurations and complex component interactions. Nevertheless, understanding specification costs in large industrial systems requires longitudinal studies. The second threat regards sample size. While sufficient for identifying major usability issues through qualitative analysis, the sample may lack statistical significance. This is acceptable given that the primary goal is identifying usability barriers rather than measuring their prevalence. The third threat concerns generalizability to other specification languages. Some findings may be specific to ROSpec's design choices rather than challenges in robotics specifications. However, these findings provide guidance for its improvement and inform the design of future specification languages.

External Validity. I identify two primary external validity threats: participant recruitment and study environment. The first threat relates to the recruitment strategy and participant pool. Participants who respond to online recruitment or agree to participate in research studies may differ from the broader ROS developer population. They may be more motivated to adopt new tools, more comfortable with formal notations, or have more flexible schedules. Furthermore, industrial developers under tight deadlines or those skeptical of formal methods may be underrepresented. To mitigate this, I recruit from multiple channels (ROS Discourse, GitHub, social media) to capture developers with diverse backgrounds. The second threat concerns differences between the controlled study environment and real-world development contexts. The provided components (Navigation Stack 2) and case study system are likely unfamiliar to most participants, whereas real specification tasks often involve code developers wrote or maintain, where they possess deeper contextual knowledge. For component writers who specify their own components, this threat is partially mitigated as they work with familiar code. However, for integrators and the documentation translation task, unfamiliarity may inflate perceived difficulty.

6.2 Conclusion

In this chapter, I propose a usability study to understand how developers interact with specification languages for component-based robot software. Through task-based observation and semi-structured interviews with component writers and integrators, I examine the challenges developers face when creating, adapting, and reasoning about specifications for ROS components and systems. The findings from this study will provide insights to guide the design of future specification languages for robotics. By identifying usability barriers, missing language features, and workflow integration challenges, this work establishes requirements for specification languages that developers adopt and use.

Chapter 7

Proposed Timeline

7.1 Timeline

I propose the following timeline for completing the work in this proposal:

	Task	**Done**	**Fall 2025**	**Spring 2026**	**Summer 2026**
Chapter 3		🟩			
Chapter 4		🟩			
Chapter 5	Build tool		🟦		
	Evaluation			🟦	
	Paper Submission			🟦	
Chapter 6	Submit IRB		🟦		
	Conduct Study			🟦	
	Paper Submission				🟦
Dissertation	Writing				🟦
	Defense				🟦

Chapter 3 (Misconfigurations Study) and Chapter 4 (ROSpec) are complete and require no further updates. Chapter 5 tool development is currently in-progress, already being able to recover ROS API calls from the source-code, and infer configurations and architecture from documentation throughout the release history. Chapter 6 outline of the usability study and creation of the IRB is currently in-progress, expected to be submitted Fall 2025, and interviews to be conducted early 2026.

Chapter 8

Conclusion

ROS-based robot software is challenging to configure correctly. Developers frequently introduce misconfigurations when integrating reusable components, leading to unintended and potentially dangerous behavior during deployment. These misconfigurations arise because components make implicit assumptions about their configuration and integration that are neither checked nor documented. This thesis addresses the misconfiguration problem through three main contributions: understanding what misconfigurations occur in practice, providing a *usable* language to specify and detect them, and studying whether specification-driven approaches are sustainable as systems evolve.

In Chapter 3, I conducted an empirical study of ROS Answers to identify and categorize the misconfigurations developers face when building robot software systems. Through qualitative analysis of 228 questions, I derived a taxonomy of 12 high-level categories and 50 sub-categories of misconfigurations. A subsequent literature review revealed that existing detection techniques address only 23 of these 50 categories, leaving significant gaps in the detection of misconfigurations. This taxonomy provides the foundation for systematically tackling the misconfigurations by revealing which types of errors are yet to be addressed.

Building on these findings, Chapter 4 introduced ROSpec, a domain-specific language for specifying component configurations and integration in ROS systems. ROSpec addresses the misconfigurations identified in the empirical study by allowing developers to specify constraints over parameters, define dependencies between configurations, and describe communication architectures. The language uses established type system concepts while embedding ROS domain knowledge, making specifications both expressive and accessible to practitioners. I demonstrated ROSpec's abilities by specifying a medium-sized warehouse robot system and writing partial specifications for misconfigured components from the ROS Answers dataset.

However, specification languages only provide value if they remain synchronized with evolving implementations and are usable by practitioners in practice. Chapter 5 proposes a large-scale empirical study to understand how ROS architectures evolve and how documentation keeps pace with these changes. By analyzing architectural evolution across releases and measuring documentation drift, this work will establish whether specification-driven approaches are sustainable as systems grow in complexity. The proposed study will reveal which architectural elements change most frequently and are most prone to drift, informing both the design of automated drift detection tools and documentation maintenance practices.

Chapter 6 proposes a usability study to evaluate how developers interact with specification languages in practice. Through task-based observation with component writers and integrators, this work will identify the challenges developers face when creating, adapting, and reasoning about specifications. These findings will provide insights to guide the improvement of ROSpec and inform the design of future specification languages that developers will actually adopt.

Together, these contributions establish the foundation for addressing misconfigurations in ROS-based robot software. The empirical study reveals what problems exist, the specification language provides a means to detect them, the drift study establishes whether specifications can be maintained over time, and the usability study ensures specifications are practical for developers. By understanding misconfigurations in ROS-based robot software, we can develop a usable domain-specific language for specifying component configurations and their integration to help detect real-world misconfigurations prior to deployment.

Bibliography

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *International Conference on Software Testing, Validation and Verification*, pages 96–107, 2020. doi: 10.1109/ICST46399.2020.00020. 1, 2.2, 4.1, 6

[2] Nicholas Albergo, Vivek Rathi, and John-Paul Ore. Understanding xacro misunderstandings. In *International Conference on Robotics and Automation*, pages 6247–6252, 2022. doi: 10.1109/ICRA46639.2022.9812349. 1, 3.1.1, 3.3, 4.1

[3] Michel Albonico, Ivano Malavolta, Gustavo Pinto, Emitza Guzman, Katerina Chinnappan, and Patricia Lago. Mining energy-related practices in robotics software. In *International Conference on Mining Software Repositories*, pages 483–494, 2021. doi: 10.1109/MSR52588.2021.00060. 3.1.1

[4] Michel Albonico, Milica Dordevic, Engel Hamer, and Ivano Malavolta. Software engineering research on the robot operating system: A systematic mapping study. *Journal of Systems and Software*, 197:111574, 2023. doi: 10.1016/J.JSS.2022.111574. 3.2

[5] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon Univeristy School of Computer Science, 1997. Issued as CMU Technical Report CMU-CS-97-144. 4.2

[6] Henrik Andreasson, Giorgio Grisetti, Todor Stoyanov, and Alberto Pretto. *Software Architectures for Mobile Robots*, pages 1–11. Springer Berlin Heidelberg, 2020. ISBN 978-3-642-41610-1. doi: 10.1007/978-3-642-41610-1_160-1. 2

[7] Boaz Arad, Jos Balendonck, Ruud Barth, Ohad Ben-Shahar, Yael Edan, Thomas Hellström, Jochen Hemming, Polina Kurtser, Ola Ringdahl, Toon ielen, and Bart van Tuijl. Development of a sweet pepper harvesting robot. *Journal of Field Robotics*, 37 (6):1027–1039, 2020. 1

[8] Anonymous Author(s). Standing on the shoulders of giants: An empirical study on ros-based open source systems. In *International Conference on Mining Software Repositories*, 2026. 5.1.1

[9] Samira Badrloo, Masood Varshosaz, Saied Pirasteh, and Jonathan Li. Image-based obstacle detection methods for the safe navigation of unmanned vehicles: A review. *Remote Sensing*, 14(15):3824, 2022. doi: 10.3390/RS14153824. 1

[10] Gianluca Bardaro, Andrea Semprebon, and Matteo Matteucci. A use case in model-

based robot development using AADL and ROS. In *International Workshop on Robotics Software Engineering*, pages 9–16, 2018. doi: 10.1145/3196558.3196560. 5

[11] Gianluca Bardaro, Andrea Semprebon, Agnese Chiatti, and Matteo Matteucci. From models to software through automatic transformations: An AADL to ROS end-to-end toolchain. In *International Conference on Robotic Computing*, pages 580–585, 2019. doi: 10.1109/IRC.2019.00118. 4.2, 5, 6

[12] BCC Research. Robotics: Technologies and global markets. https://www.bccresearch.com/market-research/engineering/robotics.html, 2023. Accessed: 2025-09-11. 1

[13] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. A systematic review of cloud modeling languages. *ACM Computing Surveys*, pages 22:1–22:38, 2018. doi: 10.1145/3150227. 4

[14] Gérard Berry. The foundations of esterel. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 425–454, 2000. 4.2

[15] Herman Bruyninckx. Open robot control software: the OROCOS project. In *International Conference on Robotics and Automation*, pages 2523–2528, 2001. doi: 10.1109/ROBOT.2001.933002. 2

[16] Loli Burgueño, Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. Using physical quantities in robot software models. In *International Workshop on Robotics Software Engineering*, pages 23–28, 2018. doi: 10.1145/3196558.3196562. 3.1, 3.2.3

[17] Ricardo Caldas, Juan Antonio Piñera García, Matei Schiopu, Patrizio Pelliccione, Genaína Rodrigues, and Thorsten Berger. Runtime verification and field-based testing for ros-based robotic systems. *IEEE Transactions on Software Engineering*, 50(10): 2544–2567, 2024. doi: 10.1109/TSE.2024.3444697. 4.7

[18] Javier Cámara, Bradley R. Schmerl, and David Garlan. Software architecture and task plan co-adaptation for mobile service robots. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 125–136, 2020. doi: 10.1145/3387939.3391591. 4.2

[19] Paulo Canelas. Artifact for "understanding misconfigurations in ros: An empirical study and current approaches", July 2024. URL https://doi.org/10.5281/zenodo.12643079. 3

[20] Paulo Canelas, Miguel Tavares, Ricardo Cordeiro, Alcides Fonseca, and Christopher Steven Timperley. An experience report on challenges in learning the robot operating system. In *International Workshop on Robotics Software Engineering*, pages 33–38, 2022. doi: 10.1145/3526071.3527521. 4.1, 4.6.1

[21] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. Understanding misconfigurations in ros: An empirical study and current approaches. In *International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680350. 1.2, 4, 4.2, 4.3.1, 4.4.1, 4.5.1, 4.6.2, 4.7, 5, 5.1

[22] Paulo Canelas, Trenton Tabor, John-Paul Ore, Alcides Fonseca, Claire Le Goues, and Christopher Steven Timperley. Is it a bug? understanding physical unit mismatches in robot software. In *International Conference on Robotics and Automation*, pages 1–7, 2024. doi: 10.1109/ICRA57147.2024.10611413. 3.3, 4.4.2

[23] Paulo Canelas, Bradley Schmerl, Alcides Fonesca, and Christopher S. Timperley. The usability argument for ros-based robot architectural description languages. In *Annual Workshop On The Intersection of HCI and PL*, 2025. doi: 10.1184/R1/29087072.v1. 4.2

[24] Paulo Canelas, Bradley Schmerl, Alcides Fonesca, and Christopher S. Timperley. Rospec: A domain-specific language for ros-based robot software. In *Proceedings of the ACM on Programming Languages*, 2025. 1.2, 6

[25] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ROS applications. In *International Conference on Intelligent Robots and Systems*, pages 7249–7254, 2020. doi: 10.1109/IROS45743.2020.9341085. 3.1

[26] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. Test-case prioritization for configuration testing. In Cristian Cadar and Xiangyu Zhang, editors, *International Symposium on Software Testing and Analysis*, pages 452–465, 2021. doi: 10.1145/3460319.3464810. 4.7

[27] Sachin Chitta, Ioan Alexandru Sucan, and Steve Cousins. Moveit! [ROS topics]. *Robotics & Automation Magazine*, 19(1):18–19, 2012. 1, 4.1, 1

[28] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ROS. *The Journal of Open Source Software*, 2:456, 2017. doi: 10.21105/JOSS.00456. 2.2, 4.1

[29] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, Chen Li, Franziska Meier, Dan Negrut, Ludovic Righetti, Alberto Rodriguez, Jie Tan, and Jeff Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1), 2021. ISSN 0027-8424. doi: 10.1073/pnas.1907856118. URL https://www.pnas.org/content/118/1/e1907856118. 1

[30] David Coleman, Ioan A. Șucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *Journal of Software Engineering for Robotics*, 5(1):3–16, May 2014. URL http://arxiv.org/abs/1404.3785. 1, 2.2, 3.1.3, 4.1

[31] Andrei Cramariuc, Aleksandar Petrov, Rohit Suri, Mayank Mittal, Roland Siegwart, and Cesar Cadena. Learning camera miscalibration detection. In *International Conference on Robotics and Automation*, pages 4997–5003, 2020. doi: 10.1109/ICRA40945.2020.9197378. 3.1

[32] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John H. Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew Wells. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):670–697, 2024. doi: 10.1145/3649835. 4.3.1

[33] Sandipan Das, Navid Mahabadi, Addi Djikic, Cesar Nassir, Saikat Chatterjee, and Maurice F. Fallon. Extrinsic calibration and verification of multiple non-overlapping field of view lidar sensors. In *International Conference on Robotics and Automation*, pages 919–925, 2022. URL https://doi.org/10.1109/ICRA46639.2022.9811704. 3.1

[34] João Pedro Vieira David. Improving machine learning pipeline creation using visual programming and static analysis. Master's thesis, Universidade de Lisboa, 2021. 4.7

[35] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24. 4.5.2

[36] Didier Delanote, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Using aadl in model driven development. In *International Workshop on UML and AADL*, pages 1–10, 2007. 4.2

[37] Byron DeVries, Erik M. Fredericks, and Betty H. C. Cheng. Analysis and monitoring of cyber-physical systems via environmental domain knowledge & modeling. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 11–17, 2021. doi: 10.1109/SEAMS51251.2021.00013. 3.1

[38] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160, 2012. ISBN 9783642343261. doi: 10.1007/978-3-642-34327-8_16. 1, 4, 4.2

[39] Tobias Dürschmid, Christopher S. Timperley, David Garlan, and Claire Le Goues. Rosinfer: Statically inferring behavioral component models for ros-based robotics systems. In *International Conference on Software Engineering*, 2024. doi: 10.1145/3597503.3639206. 3.3

[40] Tobias Dürschmid, Christopher Steven Timperley, David Garlan, and Claire Le Goues. Rosinfer: Statically inferring behavioral component models for ros-based robotics systems. In *International Conference on Software Engineering*, pages 144:1–144:13, 2024. doi: 10.1145/3597503.3639206. 1, 4.2, 4.7, 4.7, 5.1, 5.1.1

[41] Birhanu Eshete, Adolfo Villafiorita, and Komminist Weldemariam. Early detection of security misconfiguration vulnerabilities in web applications. In *International Conference on Availability, Reliability and Security*, pages 169–174, 2011. doi: 10.1109/ARES.2011.31. 3

[42] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. The robot

operating system: Package reuse and community dynamics. *Journal of Systems and Software*, pages 226–242, 2019. doi: 10.1016/J.JSS.2019.02.024. 1, 2.2, 3, 3.1, 4.1

[43] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *Computing Surveys*, 35(2):114–131, jun 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. 2.1

[44] Douglas Ezzy. *Qualitative Analysis*. Routledge, 2013. doi: 10.4324/9781315015484. 3.1.1

[45] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012. ISBN 978-0-321-88894-5. 4.2

[46] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. Rosmonitoring: A runtime verification framework for ROS. In *Towards Autonomous Robotic Systems*, volume 12228 of *Lecture Notes in Computer Science*, pages 387–399. Springer, 2020. doi: 10.1007/978-3-030-63486-5_40. 4.7

[47] Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wasowski. The forgotten case of the dependency bugs: on the example of the robot operating system. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 21–30, 2020. doi: 10.1145/3377813.3381364. 1, 3.3

[48] Alcides Fonseca, Paulo Santos, and Sara Silva. The usability argument for refinement typed genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 18–32. Springer, 2020. doi: 10.1007/978-3-030-58115-2_2. 4.2

[49] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. A domain specific language for kinematic models and fast implementations of robot dynamics algorithms, 2013. 1, 4, 4.2

[50] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *International Conference on Software Engineering*, pages 1520–1532, 2023. doi: 10.1109/ICSE48619.2023.00132. 4.2, 4.4.1

[51] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. Usability barriers for liquid types. In *Conference on Programming Language Design and Implementation*, 2025. doi: 10.1145/3729327. 4.7

[52] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *International Conference on Software Engineering*, pages 385–396, 2020. doi: 10.1145/3377811.3380397. 1, 3.3

[53] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: a perspective from the service robotics domain. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 593–604, 2020. doi: 10.1145/3368089.3409743. 1, 3.3

[54] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description

of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000. 4.2

[55] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66–69, 2009. doi: 10.1109/MS.2009.86. 3.1.1

[56] Marc Geilen and Twan Basten. Requirements on the execution of kahn process networks. In Pierpaolo Degano, editor, *Programming Languages and Systems, European Symposium on Programming*, volume 2618, pages 319–334, 2003. doi: 10.1007/3-540-36575-3_22. 4.2

[57] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Aldine, 2017. doi: 10.4324/9780203793206. 3.1.1

[58] Nicolas Gobillot, Charles Lesire, and David Doose. A modeling framework for software architecture specification and validation. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 303–314, 2014. doi: 10.1007/978-3-319-11900-7_26. 1, 4, 4.2

[59] Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. Formal specifications from natural language, 2022. URL https://arxiv.org/abs/2206.01962. 4.7

[60] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 2002. 4.2

[61] Ruidong Han, Chao Yang, Siqi Ma, Jianfeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search. In *International Conference on Software Engineering*, pages 462–473, 2022. doi: 10.1145/3510003.3510084. 3.1

[62] Najmul Hassan, Saira Andleeb Gillani, Ejaz Ahmed, Ibrar Yaqoob, and Muhammad Imran. The role of edge computing in internet of things. *IEEE Communications Magazine*, 56(11):110–115, 2018. doi: 10.1109/MCOM.2018.1700906. 4.7

[63] Eric Heiden, Christopher E. Denniston, David Millard, Fabio Ramos, and Gaurav S. Sukhatme. Probabilistic inference of simulation parameters via parallel differentiable simulation. In *International Conference on Robotics and Automation*, pages 3638–3645, 2022. doi: 10.1109/ICRA46639.2022.9812293. 3.1, 3.2.3

[64] Md. Abir Hossen, Sonam Kharade, Bradley R. Schmerl, Javier Cámara, Jason M. O'Kane, Ellen C. Czaplinski, Katherine A. Dzurilla, David Garlan, and Pooyan Jamshidi. Care: Finding root causes of configuration issues in highly-configurable robots. *CoRR*, abs/2301.07690, 2023. 4.7

[65] Md. Abir Hossen, Sonam Kharade, Jason M. O'Kane, Bradley R. Schmerl, David Garlan, and Pooyan Jamshidi. CURE: simulation-augmented auto-tuning in robotics. *CoRR*, abs/2402.05399, 2024. doi: 10.48550/ARXIV.2402.05399. 4.7

[66] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. ROSRV: runtime verification for robots. In *Runtime Verification - 5th International Conference*, volume 8734 of *Lecture Notes in Computer Science*, pages 247–254, 2014. 4.7

[67] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. Confvalley: a systematic configuration validation framework for cloud services. In *European Conference on Computer Systems*, pages 19:1–19:16, 2015. doi: 10.1145/2741948.2741963. 3

[68] Yuan Huang, Yinan Chen, Xiangping Chen, and Xiaocong Zhou. Are your comments outdated? toward automatically detecting code-comment consistency. *Journal of Software: Evolution and Process*, 37(1), 2025. doi: 10.1002/SMR.2718. 1

[69] Sajjad Hussain. Investigating architecture description languages (adls) a systematic literature review. Master's thesis, Linköping University, Software and Systems, The Institute of Technology, 2013. 1, 4, 6

[70] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer mistakes in writing android manifests: an empirical study of configuration errors. In *International Conference on Mining Software Repositories*, pages 25–36, 2017. doi: 10.1109/MSR.2017.41. 3.3

[71] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian G. Elbaum, and Yonghwi Kwon. Swarmbug: debugging configuration bugs in swarm robotics. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 868–880, 2021. doi: 10.1145/3468264.3468601. 3.1, 3.4

[72] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian G. Elbaum, and Zhaogui Xu. Phys: probabilistic physical unit assignment and inconsistency detection. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 563–573, 2018. doi: 10.1145/3236024.3236035. 1, 3, 3.1, 3.2.3, 3.4, 4.2

[73] Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian G. Elbaum. PHYSFRAME: type checking physical frames of reference for robotic systems. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 45–56, 2021. doi: 10.1145/3468264.3468608. 3, 3.1

[74] Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. A lightweight polyglot code transformation language. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1288–1312, 2024. doi: 10.1145/3656429. 3.4, 5.2.1

[75] Seulbae Kim and Taesoo Kim. Robofuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 447–458, 2022. doi: 10.1145/3540250.3549164. 3.1, 3.4

[76] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. Liquid resource types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):

106:1–106:29, 2020. doi: 10.1145/3408988. 4.2

[77] Sophia Kolak, Afsoon Afzal, Michael Hilton, Claire Le Goues, and Christopher S Timperley. It takes a village to build a robot: An empirical study of the ros ecosystem. In *International Conference on Software Maintenance and Evolution*, pages 430–440, 2020. doi: 10.1109/ICSME46990.2020.00048. 2.2

[78] Sophia Kolak, Afsoon Afzal, Michael Hilton, Claire Le Goues, and Christopher S Timperley. It takes a village to build a robot: An empirical study of the ros ecosystem. In *International Conference on Software Maintenance and Evolution*, pages 430–440, 2020. 1

[79] David Kolevski and Katina Michael. Edge computing and iot data breaches: Security, privacy, trust, and regulation. *IEEE Technology and Society Magazine*, 43(1):22–32, 2024. doi: 10.1109/MTS.2024.3372605. 4.7

[80] Sitar Kortik and Tejas Kumar Shastha. Formal verification of ROS based systems using a linear logic theorem prover. In *International Conference on Robotics and Automation*, pages 9368–9374, 2021. doi: 10.1109/ICRA48506.2021.9561191. 3.1

[81] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.11.001. 3.4

[82] Anit Kumar and Dhanpratap Singh. Securing iot devices in edge computing through reinforcement learning. *Computers and Security*, 155:104474, 2025. doi: 10.1016/J.COSE.2025.104474. 4.7

[83] William Landi. Undecidability of static analysis. *Letters Programming Language Systems*, 1(4):323–337, dec 1992. ISSN 1057-4514. doi: 10.1145/161494.161501. 3.4

[84] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. {STORM}: Refinement types for secure web applications. In *{USENIX} Symposium on Operating Systems Design and Implementation)*, pages 441–459, 2021. 4.2

[85] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM Programming Languages*, 7(PLDI):1533–1557, 2023. URL https://doi.org/10.1145/3591283. 4.2, 4.4.1

[86] Luka Lelovic, Michael Mathews, Amr S. Abdelfattah, and Tomás Cerný. Microservices architecture language for describing service view. In *International Conference on Cloud Computing and Services Science*, pages 220–227, 2023. doi: 10.5220/0011850200003488. 4.7

[87] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. Large language models as configuration validators. In *International Conference on Software Engineering*, pages 204–216, 2024. 4.7

[88] Xiangke Liao, Shulin Zhou, Shanshan Li, Zhouyang Jia, Xiaodong Liu, and Haochen He. Do you really know how to configure your software? configuration constraints in source code may help. *Transactions on Reliability*, 67(3):832–846, 2018. doi:

10.1109/TR.2018.2834419. 3

[89] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems*, 20(4):36:1–36:27, 2021. doi: 10.1145/3448128. 4.2

[90] Markus Look, Antonio Navarro Perez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Black-box integration of heterogeneous modeling languages for cyber-physical systems, 2014. 1, 4, 4.2

[91] Markus Luckey and Gregor Engels. High-quality specification of self-adaptive software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 143–152, 2013. doi: 10.1109/SEAMS.2013.6595501. 4.2

[92] Steve Macenski, Tom Moore, David V. Lu, Alexey Merzlyakov, and Michael Ferguson. From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2. *Robotics Autonomous Systems*, 168: 104493, 2023. doi: 10.1016/J.ROBOT.2023.104493. 2.2

[93] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022. doi: 10.1126/SCIROBOTICS.ABM6074. 3.1.2, 4.1, 4.3.1

[94] Steven Macenski, Tully Foote, Brian P. Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022. doi: 10.1126/SCIROBOTICS.ABM6074. 1, 2.2

[95] Ratul Mahajan, David Wetherall, and Thomas E. Anderson. Understanding BGP misconfiguration. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–16, 2002. doi: 10.1145/633025. 633027. 3

[96] Jabier Martinez, Alejandra Ruiz, Ansgar Radermacher, and Stefano Tonetta. Assumptions and guarantees for composable models in papyrus for robotics. In *International Workshop on Robotics Software Engineering*, pages 1–4, 2021. doi: 10.1109/RoSE52553.2021.00007. 1, 4, 4.2

[97] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Transactions of Software Engineering*, 26(1):70–93, 2000. doi: 10.1109/32.825767. 1, 4, 6

[98] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006. doi: 10.5772/5761. 2

[99] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (CARMEN) toolkit. In *International Conference on Intelligent Robots and Systems*, pages 2436–2441. IEEE, 2003. doi: 10.1109/IROS.2003.1249235. 2

[100] T. Moore and D. Stouch. A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In *International Conference on Intelligent Autonomous*

Systems, pages 335–348, 2014. doi: 10.1007/978-3-319-08338-4_25. 2.2

[101] Henry Muccini and Mohammad Sharaf. Caps: Architecture description of situational aware cyber physical systems. In *International Conference on Software Architecture*, pages 211–220, 2017. doi: 10.1109/ICSA.2017.21. 1, 4, 4.2

[102] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering in Robotics*, 7(1):75–99, 2016. 4, 4.2

[103] Nuro. Introducing our next-generation nuro, Jan 2022. URL https://medium.com/nuro/introducing-our-next-generation-nuro-8c1c63488342. 1

[104] John-Paul Ore, Sebastian G. Elbaum, and Carrick Detweiler. Dimensional inconsistencies in code and ROS messages: A study of 5.9m lines of code. In *International Conference on Intelligent Robots and Systems*, pages 712–718, 2017. doi: 10.1109/IROS.2017.8202229. 1, 3.3, 4.1

[105] Francisco J. Ortiz, Diego Alonso, Francisca Rosique, Francisco Sánchez-Ledesma, and Juan Angel Pastor. A component-based meta-model and framework in the model driven toolchain c-forge. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 340–351, 2014. doi: 10.1007/978-3-319-11900-7_29. 1, 4, 4.2

[106] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Exploring New Frontiers of Theoretical Informatics, International Conference on Theoretical Computer Science*, pages 437–450, 2004. doi: 10.1007/1-4020-8141-3_34. 4.5.2

[107] Mert Ozkaya and Christos Kloukinas. Are we there yet? analyzing architecture description languages for formal analysis, usability, and realizability. In *Euromicro Conference on Software Engineering and Advanced Applications*, pages 177–184, 2013. doi: 10.1109/SEAA.2013.34. 4.2

[108] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992. doi: 10.1145/141874.141884. 5

[109] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Conference on Programming Language Design and Implementation*, pages 522–538, 2016. doi: 10.1145/2908080.2908093. 4.2

[110] Patrick Probst. A review of the role of robotics in surgery: To davinci and beyond! *Missouri Medicine*, pages 389–396, 2023. ISSN 0026-6620. 1

[111] Proteus. I'm amazon's first autonomous robot. follow me around on my typical workday (watercooler break included)., 10 2024. URL https://www.aboutamazon.com/news/operations/amazon-robotics-autonomous-robot-proteus-warehouse-packages. 4.6.1

[112] Morgan Quigley, Ken Conle, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3(3.2):1–6, 01 2009. 1, 2

[113] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita.

Security misconfigurations in open source kubernetes manifests: An empirical study. *Transactions on Software Engineering and Methodology*, 32(4):99:1–99:36, 2023. doi: 10.1145/3579639. 3

[114] Akshay Rajhans, Shang-Wen Cheng, Bradley Schmerl, David Garlan, Bruce Krogh, Clarence Agbi, and Ajinkya Bhave. An architectural approach to the design and analysis of cyber-physical systems. *Electronic Communication of the European Association of Software Science and Technology*, 2009. doi: 10.14279/tuj.eceasst.21.286. 1, 4, 4.2

[115] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. An extensible model-based framework for robotics software development. In *International Conference on Robotic Computing*, pages 73–76, 2017. doi: 10.1109/IRC.2017.21. 4.2

[116] Nicholas Rescher and Alasdair Urquhart. *Temporal logic*, volume 3. Springer Science & Business Media, 2012. 4.7

[117] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Conference on Programming Language Design and Implementation*, pages 159–169. ACM, 2008. doi: 10.1145/1375581.1375602. 4, 4.2, 4.4.1, 4.5.1, 4.5.2

[118] ROS-Industrial. Current members - ros-industrial, Nov 2021. URL https://rosindustrial.org/ric/current-members/. 1, 2

[119] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1):63–86, 2011. doi: 10.1002/SPE.999. 1

[120] Ivan Ruchkin, Bradley R. Schmerl, and David Garlan. Architectural abstractions for hybrid programs. In *Symposium on Component-Based Software Engineering*, pages 65–74, 2015. doi: 10.1145/2737166.2737167. 4.2

[121] André Santos, Alcino Cunha, and Nuno Macedo. The high-assurance ROS framework. URL https://github.com/git-afsantos/haros. 5

[122] André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. Mining the usage patterns of ROS primitives. In *International Conference on Intelligent Robots and Systems*, pages 3855–3860, 2017. doi: 10.1109/IROS.2017.8206237. 2.1, 5.2.1

[123] André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. Mining the usage patterns of ROS primitives. In *International Conference on Intelligent Robots and Systems*, pages 3855–3860, 2017. doi: 10.1109/IROS.2017.8206237. 4.1, 4.3.1, 4.6.2

[124] André Santos, Alcino Cunha, and Nuno Macedo. Property-based testing for the robot operating system. In *International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 56–62, 2018. doi: 10.1145/3278186.3278195. 3.2.1, 3.4

[125] André Santos, Alcino Cunha, and Nuno Macedo. The high-assurance ROS framework.

In *International Workshop on Robotics Software Engineering*, pages 37–40, 2021. doi: 10.1109/ROSE52553.2021.00013. 1, 3.2.1, 3.1, 4.2, 4.7, 4.7, 6

[126] Amita Singh, Fabian Quint, Patrick Bertram, and Martin Ruskowski. A framework for semantic description and interoperability across cyber-physical systems. *International Journal on Advances in Intelligent Systems*, pages 70–81, 2019. 1, 4, 4.2

[127] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, 2019. ISSN 0148-2963. doi: 10.1016/j.jbusres.2019.07.039. 3.2

[128] Wen Siang Tan, Markus Wagner, and Christoph Treude. Detecting outdated code element references in software repository documentation. *Empirical Software Engineering*, 29(1):5, 2024. doi: 10.1007/S10664-023-10397-6. 1

[129] Max Taylor, Johnathon Aurand, Feng Qin, Xiaorui Wang, Brandon Henry, and Xiangyu Zhang. Sa4u: Practical static analysis for unit type error detection. In *International Conference on Automated Software Engineering*, 2023. ISBN 9781450394758. doi: 10.1145/3551349.3556937. 3.1, 3.2.3, 3.4, 3.4

[130] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. How developers discuss architecture smells? an exploratory study on stack overflow. In *International Conference on Software Architecture*, pages 91–100, 2019. doi: 10.1109/ICSA.2019.00018. 3.1.1

[131] Christopher Timperley and A Wąsowski. 188 ROS bugs later: Where do we go from here? *ROSCON'19*, 2019. URL https://roscon.ros.org/2019/talks/roscon2019_188_bugs_later.pdf. 1, 3.3

[132] Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wasowski. Robust: 221 bugs in the robot operating system. *Empirical Software Engineering*, 29(3):57, 2024. doi: 10.1007/S10664-024-10440-0. 1, 3.3, 4.1

[133] Christopher Steven Timperley, Tobias Dürschmid, Bradley R. Schmerl, David Garlan, and Claire Le Goues. Rosdiscover: Statically detecting run-time architecture misconfigurations in robotics systems. In *International Conference on Software Architecture*, pages 112–123, 2022. doi: 10.1109/ICSA53651.2022.00019. 1, 3, 3.1, 3.3, 3.4, 4.2, 4.3.2, 4.7, 4.7, 5.1, 5.1.1, 5.2.1

[134] Daniella Tola and Peter Corke. Understanding URDF: A survey based on user experience. In *International Conference on Automation Science and Engineering*, pages 1–7, 2023. doi: 10.1109/CASE56687.2023.10260660. 4.1

[135] Sabrina Tseng, Erik Hemberg, and Una-May O'Reilly. Synthesizing programs from program pieces using genetic programming and refinement type checking. In *European Conference on Genetic Programming*, pages 197–211, 2022. doi: 10.1007/978-3-031-02056-8_13. 4.2

[136] Aaron Turon. Rust's language ergonomics initiative, 3 2017. URL https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html. 4.7

[137] Tetsuya Uchiumi, Shinji Kikuchi, and Yasuhide Matsumoto. Misconfiguration detection for cloud datacenters using decision tree analysis. In *Asia-Pacific Network Operations and Management Symposium*, pages 1–4, 2012. doi: 10.1109/APNOMS.2012.6356072. 3

[138] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In *International Conference on Functional Programming*, pages 269–282. ACM, 2014. doi: 10.1145/2628136.2628161. 4.2, 4.4.1

[139] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Conference on Programming Language Design and Implementation*, pages 310–325. ACM, 2016. doi: 10.1145/2908080.2908110. 4.2

[140] Kaiyu Wan, Ka Lok Man, and Danny Hughes. Specification, analyzing challenges and approaches for cyber-physical systems (cps). *Engineering Letters*, 2010. 4, 4.2

[141] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 65030–65055, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf. 4.7

[142] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 20–31, 2021. doi: 10.1145/3468264.3468559. 1, 3.3

[143] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, (POPL):63:1–63:30, 2018. doi: 10.1145/3158151. 4.2

[144] Hongxing Wei, Xinming Duan, Shiyi Li, Guofeng Tong, and Tianmiao Wang. A component based design framework for robot software architecture. In *International Conference on Intelligent Robots and Systems*, pages 3429–3434, 2009. doi: 10.1109/IROS.2009.5354161. 1, 4, 4.2

[145] Dennis Leroy Wigand, Niels Dehio, and Sebastian Wrede. Model-based specification of control architectures for compliant interaction with the environment. In *International Conference on Intelligent Robots and Systems*, pages 7241–7248, 2020. doi: 10.1109/IROS45743.2020.9340718. 3.1

[146] Thomas Witte and Matthias Tichy. Checking consistency of robot software architectures in ROS. In *Workshop on Robotics Software Engineering*, pages 1–8, 2018. doi: 10.1145/3196558.3196559. 3.1

[147] Eoin Woods and Rich Hilliard. Architecture description languages in practice session report. In *Working Conference on Software Architecture*, pages 243–246, 2005. doi: 10.1109/WICSA.2005.15. 1, 4, 4.7, 6

[148] Valentin Wüest, Vijay Kumar, and Giuseppe Loianno. Online estimation of geometric and inertia parameters for multirotor aerial vehicles. In *International Conference on*

Robotics and Automation, pages 1884–1890, 2019. doi: 10.1109/ICRA.2019.8794274. 3.1, 3.2.3

[149] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Andrew W. Appel and Alex Aiken, editors, *Symposium on Principles of Programming Languages*, pages 214–227, 1999. doi: 10.1145/292540.292560. 4

[150] Jingtao Xia, Junrui Liu, Nicholas Brown, Yanju Chen, and Yu Feng. Refinement types for visualization. In *International Conference on Automated Software Engineering*, pages 1871–1881. ACM, 2024. doi: 10.1145/3691620.3695550. 4.2

[151] Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *Computing Surveys*, 47(4):70:1–70:41, 2015. doi: 10.1145/2791577. 3.3

[152] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng and† Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Symposium on Operating Systems Principles*, pages 244–259, 2013. doi: 10.1145/2517349.2522727. 3.3

[153] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, pages 159–172, 2011. doi: 10.1145/2043556.2043572. 3.3

[154] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, pages 159–172, 2011. doi: 10.1145/2043556.2043572. 3

[155] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021. doi: 10.1145/3485517. 4.7

[156] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: exploiting system environment and correlation information for misconfiguration detection. In *Architectural Support for Programming Languages and Operating Systems*, pages 687–700, 2014. doi: 10.1145/2541940.2541983. 3

[157] Kaiyu Zheng. ROS navigation tuning guide. *CoRR*, abs/1706.09068, 2017. URL http://arxiv.org/abs/1706.09068. 1

Complete Language Semantics

.1 Grammar

$$
\begin{array}{rrl}
\text{Types} & T ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{string} \mid \texttt{t} \\
& & \mid T[\,] \quad \mid \textbf{Enum}[\overline{x}] \\
& & \mid \textbf{struct}\ \ \{\ \overline{x : T}\ \} \\
& & \mid x : T\ \ \textbf{where}\ \ \{\ e\ \} \\
& & \mid \textbf{Optional}(T,\ e) \\
& & \mid \textbf{NodeT}(\overline{S_p}; \overline{S_c}; \overline{S_{f_r}}) \\
& & \mid \textbf{PluginT}(\overline{S_c}; \overline{S_p}; \overline{S_r}) \\
\text{Definitions} & D ::= & \textbf{node type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \} \\
& & \mid \textbf{node type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \}\ \ \textbf{where}\ \ \{\ e\ \} \\
& & \mid \textbf{node instance}\ \ x_1 :\ x_2\ \{\ \overline{S_{p_i}};\ \overline{S_r}\ \} \\
& & \mid \textbf{plugin type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \} \\
& & \mid \textbf{plugin type}\ \ x\ \{\ \overline{S_{p_d}};\ \overline{S_c};\ \overline{S_{f_r}}\ \}\ \ \textbf{where}\ \ \{\ e\ \} \\
& & \mid \textbf{plugin instance}\ \ x_1 :\ x_2\ \{\ \overline{S_{p_i}};\ \overline{S_r}\ \} \\
& & \mid \textbf{policy instance}\ \ x_1 :\ x_2\ \{\ \overline{S_{p_i}}\ \} \\
& & \mid \textbf{type alias}\ \ t :\ T; \\
& & \mid \textbf{message alias}\ \ t :\ T\ \{\ \overline{S_f}\ \} \\
& & \mid \textbf{message alias}\ \ t :\ T\ \{\ \overline{S_f}\ \}\ \ \textbf{where}\ \ \{\ e\ \} \\
\text{Statements} & S_{p_d} ::= & \textbf{param}\ \ x :\ T; \\
& & \mid \textbf{argument}\ \ x :\ T; \\
& & \mid \textbf{context}\ \ x :\ T; \\
& & \mid \textbf{optional param}\ \ x :\ T\ =\ e; \\
& & \mid \textbf{optional argument}\ \ x :\ T\ =\ e; \\
& S_{p_i} ::= & \textbf{param}\ \ x\ =\ e; \\
& & \mid \textbf{argument}\ \ x :\ T\ =\ e; \\
& & \mid \textbf{context}\ \ x :\ T\ =\ e; \\
& S_f ::= & \textbf{field}\ \ x : T; \\
& S_c ::= & x_1\ \ \textbf{publishes to}\ \ x_2 :\ T;
\end{array}
$$

$$\begin{array}{rl}
& \mid x_1 \;\; \texttt{subscribes to} \;\; x_2 : \; T; \\
& \mid x_1 \;\; \texttt{provides/consumes action} \;\; x_2 : \; T; \\
& \mid x_1 \;\; \texttt{provides/consumes service} \;\; x_2 : \; T; \\
& \mid S_c \;\; \textbf{with qos}(e); \\
& \mid \texttt{system}\{\; \overline{D} \;\} \\
S_{f_r} ::= & x_1 \;\; \texttt{broadcast/listens} \;\; x_2 \;\; \texttt{to} \;\; x_3; \\
S_r ::= & x_1 \;\; \texttt{remap} \;\; x_2 \;\; \texttt{to} \;\; x_3; \\
\text{Expressions} \quad e ::= & n \mid s \mid f \mid x \mid \texttt{true} \mid \texttt{false} \\
& \mid \; e < e \;\mid\; e == e \;\mid\; e \neq e \;\mid\; e \to e \;\mid\; e \wedge e \;\mid\; e \vee e \;\mid\; \neg e \\
& \mid \; x(e) \;\mid\; t\{\; \overline{x : T} \;\} \\
\text{Context} \quad \Gamma ::= & \epsilon \;\mid\; \Gamma,\; x :\; T \;\mid\; \Gamma,\; t =\; T \;\mid\; \Gamma,\; x \;\mapsto\; \overline{S_c}
\end{array}$$

.2 Formation Rules

$$\textsc{t-Int}\ \frac{}{\Gamma \vdash \texttt{int}} \qquad \textsc{t-Bool}\ \frac{}{\Gamma \vdash \texttt{bool}} \qquad \textsc{t-Float}\ \frac{}{\Gamma \vdash \texttt{float}} \qquad \textsc{t-String}\ \frac{}{\Gamma \vdash \texttt{string}} \qquad \textsc{t-Where}\ \frac{\Gamma, x : T \vdash e :\; \texttt{bool}}{\Gamma \vdash x : T \;\; \textbf{where} \;\; \{\; e \;\}}$$

$$\textsc{t-Struct}\ \frac{\Gamma \vdash \overline{T}}{\Gamma \vdash \textbf{struct} \;\; \{\; \overline{x :\; T} \;\}} \qquad \textsc{t-Optional}\ \frac{\Gamma \vdash e : U \quad \Gamma \vdash U <: T}{\Gamma \vdash \textbf{Optional}(T,\; e)} \qquad \textsc{t-NodeT}\ \frac{\forall x_i : T_i \in \overline{S_p}, \Gamma \vdash x_i : T_i}{\Gamma \vdash \text{NodeT}(\overline{S_p}; \overline{S_r})}$$

$$\textsc{t-PluginT}\ \frac{\forall x_i : T_i \in \overline{S_p}, \Gamma \vdash x_i : T_i}{\Gamma \vdash \text{PluginT}(\overline{S_p}; \overline{S_r})} \qquad \textsc{t-EnumT}\ \frac{\forall x_i \in \overline{x}, \Gamma \vdash x_i}{\Gamma \vdash \text{Enum}[\overline{x}]}$$

Figure 1: Type Formation Rules. $\boxed{\Gamma \vdash T}$

$$\textsc{C-Empty}\ \frac{}{\vdash \epsilon \text{ context}} \qquad \textsc{C-Type}\ \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash T}{\vdash \Gamma, x :\; T \text{ context}} \qquad \textsc{C-Connection}\ \frac{\vdash \Gamma \text{ context} \quad x \in \Gamma}{\vdash \Gamma, x \;\mapsto\; \overline{S_c} \text{ context}} \qquad \textsc{C-Alias}\ \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash T}{\vdash \Gamma, t \;=\; T \text{ context}}$$

Figure 2: Context Formation Rules. $\boxed{\vdash \Gamma \text{ context}}$

$$\frac{}{\Gamma \vdash b : \texttt{bool}}\ \textsc{E-Bool} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{E-Int} \qquad \frac{}{\Gamma \vdash f : \texttt{float}}\ \textsc{E-Double} \qquad \frac{}{\Gamma \vdash s : \texttt{string}}\ \textsc{E-String} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{E-Var}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1, T_2 \in \mathit{domain}(\mathbf{op})}{\Gamma \vdash e_1\ \mathbf{op}\ e_2 : \mathit{codomain}(\mathbf{op})}\ \textsc{E-ExpBinOp}$$

Figure 3: Expression Formation Rules. $\boxed{\Gamma \vdash e : T}$

$$\textsc{Sp}_{\textsc{d}}\text{-}\textsc{Param}\quad \frac{}{\Gamma \vdash \textbf{param}\ \ x:\ T \dashv x:\ T}$$

$$\textsc{Sp}_{\textsc{d}}\text{-}\textsc{Arg}\quad \frac{}{\Gamma \vdash \textbf{argument}\ \ x:\ T \dashv x:\ T}$$

$$\textsc{Sp}_{\textsc{d}}\text{-}\textsc{Ctx}\quad \frac{}{\Gamma \vdash \textbf{context}\ \ x:\ T \dashv x:\ T}$$

$$\textsc{Sf}\text{-}\textsc{Field}\quad \frac{}{\Gamma \vdash \textbf{field}\ \ x:\ T \dashv x:\ T}$$

$$\textsc{Sp}_{\textsc{d}}\text{-}\textsc{OptParam}\quad \frac{\Gamma, x : T \vdash e : U \quad \Gamma \vdash U <: T}{\Gamma \vdash \textbf{optional param}\ \ x:\ T\ =\ e \dashv x : \textbf{Optional}(T,\ e)}$$

$$\textsc{Sp}_{\textsc{d}}\text{-}\textsc{Arg}\quad \frac{\Gamma, x : T \vdash e : T \quad \Gamma \vdash U <: T}{\Gamma \vdash \textbf{optional argument}\ \ x:\ T\ =\ e \dashv x : \textbf{Optional}(T,\ e)}$$

$$\textsc{Sp}_{\textsc{i}}\text{-}\textsc{Param}\quad \frac{}{\Gamma \vdash \textbf{param}\ \ x\ =\ e \dashv x = e}$$

$$\textsc{Sp}_{\textsc{i}}\text{-}\textsc{Arg}\quad \frac{}{\Gamma \vdash \textbf{argument}\ \ x\ =\ e \dashv x = e}$$

$$\textsc{Sp}_{\textsc{i}}\text{-}\textsc{Ctx}\quad \frac{}{\Gamma \vdash \textbf{context}\ \ x\ =\ e \dashv x = e}$$

$$\textsc{Sr}\text{-}\textsc{Remap}\quad \frac{\Gamma \dashv x_1 : T \quad \Gamma \dashv x_2 : U}{\Gamma \vdash\ x_1\ \textbf{remap}\ x_2\ \textbf{to}\ x_3 \dashv \Gamma, x_3 : U}$$

$$\textsc{Sc}\text{-}\textsc{Frame}\quad \frac{x_1 \neq x_2 \quad x_2 \neq x_3}{\Gamma \vdash\ x_1\ \textbf{broadcasts}/\textbf{listens}\ x_2\ \textbf{to}\ x_3 \dashv \Gamma, x_1 \mapsto \textbf{broadcasts}/\textbf{listens}(x_2, x_3)}$$

$$\textsc{Sc}\text{-}\textsc{Publisher}\quad \frac{\Gamma \vdash x_1 : T}{\Gamma \vdash\ x_1\ \texttt{publishes to}\ \ x_2 : T \dashv \{\ \texttt{publishes to}(x_2, T)\ \}}$$

$$\textsc{Sc}\text{-}\textsc{Subscriber}\quad \frac{\Gamma \vdash x_1 : T \quad \Gamma \vdash t : T}{\Gamma \vdash\ x_1\ \texttt{subscribes to}\ \ x_2 : T \dashv \{\ \texttt{subscribes to}(x_2, T)\ \}}$$

$$\textsc{Sc}\text{-}\textsc{Service}\quad \frac{\Gamma \vdash x_1 : T \quad \Gamma \vdash x_2 : T_2}{\Gamma \vdash\ x_1\ \texttt{provides/consumes service}\ \ x_2 : T_2 \dashv \{\ \texttt{provides/consumes service}(e, T_2)\ \}}$$

$$\textsc{Sc}\text{-}\textsc{Action}\quad \frac{\Gamma \dashv x_1 : T \quad \Gamma \dashv x_2 : T_2}{\Gamma \vdash\ x_1\ \texttt{provides/consumes action}\ \ x_2 : T_2 \dashv \{\ \texttt{provides/consumes action}(x_2, T_2)\ \}}$$

Figure 4: Statement Formation Rules. $\boxed{\Gamma \vdash S_p \dashv x : T}$ $\boxed{\Gamma \vdash S_c \dashv S(x, t)}$

D-NodeType

$$\frac{}{\Gamma \vdash \textbf{node type}\ \ x\ \{\ \overline{S_p}; \overline{S_c}; \overline{S_{f_r}}\ \} \dashv \Gamma,\ x : \texttt{NodeT}(\overline{S_p}; \overline{S_c}; \overline{S_{f_r}})}$$

D-PluginType

$$\frac{}{\Gamma \vdash \textbf{plugin type}\ \ x\ \{\ \overline{S_p}; \overline{S_c}; \overline{S_{f_r}}\ \} \dashv \Gamma,\ x : \texttt{PluginT}(\overline{S_p}; \overline{S_c}; \overline{S_{f_r}})}$$

D-NodeInstance

$$\frac{\begin{array}{c}\Gamma \vdash x2 : \texttt{NodeT}(\overline{S'_p}; \overline{S_c}; \overline{S_{fr}}) \qquad \Gamma \vdash \overline{S_{p_i}} <: \overline{S'_p} \\ \overline{S'_c} = \overline{S_c} \cup \{\ \overline{S''_c} \mid [x_3 \mapsto x_4] \in \overline{S_{p_i}},\ x_4 : \texttt{PluginT}(\overline{S''_c}, \_, \_)\ \} \\ \sigma = \overline{S_{p_i}} \cup \{x \mapsto e | x : \texttt{Optional}(T, e) \in \overline{S'_p},\ x \notin \overline{S_{p_i}}\}\end{array}}{\Gamma \vdash \textbf{node instance}\ \ x_1 : x_2\ \{\ \overline{S_{p_i}};\ \overline{S_r}\ \} \dashv \Gamma,\ x_1 \mapsto S'_c[\sigma][\overline{S_r}]}$$

D-PluginInstance

$$\frac{\begin{array}{c}\Gamma \vdash x2 : \texttt{PluginT}(\overline{S'_p}; \overline{S'_c}; \overline{S_{fr}}) \qquad \Gamma \vdash \overline{S_{p_i}} <: \overline{S'_p} \\ \sigma = \overline{S_{p_i}} \cup \{x \mapsto e | x : \texttt{Optional}(T, e) \in \overline{S'_p},\ x \notin \overline{S_{p_i}}\}\end{array}}{\Gamma \vdash \textbf{plugin instance}\ \ x_1 : x_2\ \{\ \overline{S_{p_i}};\ \overline{S_r}\ \} \dashv \Gamma,\ x_1 \rightsquigarrow \texttt{PluginT}(\sigma; \overline{S'_c[\sigma][\overline{S_r}]}; \overline{S_{fr}})}$$

D-PolicyInstance

$$\frac{\Gamma \dashv \bar{[}(\overline{self(x_1 \mapsto v)})] \in S_{p_i}}{\Gamma \vdash \textbf{policy instance}\ x : qos\ \{\ \overline{S_{p_i}};\ \} \dashv \Gamma,\ x : \textbf{struct}\{\ \overline{x_1 = v}\ \}}$$

D-System

$$\frac{\begin{array}{c}\Gamma \vdash \overline{D} \dashv\ \Gamma' \\ \forall s \mapsto \textbf{subscribes to}(x,\ T_2) \in \Gamma',\ \exists\ p \mapsto \textbf{publishes to}(x,\ T_1) \in \Gamma' \wedge \Gamma' \vdash T_1 <: T_2 \\ \forall s \mapsto \textbf{publishes to}(x)\ \textbf{with}\ qos(q1) \in \Gamma', \\ \forall s' \mapsto \textbf{subscribes to}(x)\ \textbf{with}\ qos(q2) \in \Gamma',\ \texttt{check\_qos}(q1,\ q2) \\ \forall\ s \mapsto \textbf{listens}(x_1, x_2) \in \Gamma', \exists\ p \mapsto \textbf{broadcasts}(x_1, x_2) \in \Gamma' \\ \forall\ s \mapsto \textbf{broadcasts}(x_1, x_2) \in \Gamma', \forall\ s' \mapsto \textbf{broadcasts}(x_2, x_3) \in \Gamma', x_1 = x_2\end{array}}{\Gamma \vdash \textbf{system}\ \{\ \overline{D}\ \} \dashv\ \Gamma'}$$

D-TypeAlias

$$\frac{\Gamma \vdash T}{\Gamma \vdash \textbf{type alias}\ \ t : T; \dashv\ \Gamma, t : T}$$

Figure 5: Definition Formation Rules. $\boxed{\Gamma \vdash D \dashv \Gamma'}$

$$\textsc{S-Int}\ \frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \texttt{int} <: \texttt{int}} \qquad \textsc{S-Bool}\ \frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \texttt{bool} <: \texttt{bool}} \qquad \textsc{S-Float}\ \frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \texttt{float} <: \texttt{float}}$$

$$\textsc{S-String}\ \frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \texttt{string} <: \texttt{string}} \qquad \textsc{S-Var}\ \frac{t = U \in \Gamma \quad \Gamma \vdash U <: T}{\Gamma \vdash t <: T} \qquad \textsc{S-Array}\ \frac{\Gamma \vdash T <: U}{\Gamma \vdash T[] <: U[]}$$

$$\textsc{S-Struct}\ \frac{\overline{\Gamma \vdash T_i <: U_i}}{\Gamma \vdash \textbf{struct}\{\ \overline{x_i : T_i}\ \} <: \textbf{struct}\{\ \overline{x_i : U_i}\ \}} \qquad \textsc{S-Optional}\ \frac{\Gamma \vdash T <: U}{\Gamma \vdash T <: \mathit{Optional}(U, e)}$$

$$\textsc{S-Where}\ \frac{\Gamma \vdash T <: U \quad \Gamma, x_1 : T \vdash e_i \implies e_2[x_2 \mapsto x_1]}{\Gamma \vdash (x_1 : T \ \ \textbf{where}\ \ \{\ e_1\ \}) <: (x_2 : U \ \ \textbf{where}\ \ \{\ e_2\ \})}$$

$$\textsc{S-WhereL}\ \frac{\Gamma \vdash T <: U \quad \Gamma, x : T \vdash e : \texttt{bool}}{\Gamma \vdash (x : T \ \ \textbf{where}\ \ \{\ e\ \}) <: U : k} \qquad \textsc{S-WhereR}\ \frac{\Gamma \vdash T <: U \quad \Gamma, x : T \vdash e \text{ true}}{\Gamma \vdash T <: (x : U \ \ \textbf{where}\ \ \{\ e\ \}) : k}$$

$$\textsc{S-Param}\ \frac{\Gamma \vdash \mathit{self}(e) <: T}{\Gamma \vdash \textbf{param}\ x = e <: \textbf{param}\ x : T;}$$

$$\textsc{S-OptL}\ \frac{\Gamma \vdash T <: U : k}{\Gamma \vdash \textbf{Optional}(T, e) <: U : k} \qquad \textsc{S-Opt}\ \frac{\Gamma \vdash T <: U : k}{\Gamma \vdash T <: \textbf{Optional}(U, e) : k}$$

Figure 6: Subtyping Rules. $\boxed{\Gamma \vdash t_1 <: t_2 : k}$

$$\textsc{CheckQoS}\ \frac{\begin{array}{c} \Gamma \vdash \overline{x_1 : S_{p_p}} \quad \Gamma \vdash \overline{x_2 : S_{p_s}} \\ \dashv \neg(\overline{S_{p_p}}[\text{reliability}] == \text{BestEffort} \wedge \overline{S_{p_s}}[\text{reliability}] == \text{Reliable}) \\ \dashv \neg(\overline{S_{p_p}}[\text{durability}] == \text{Volatile} \wedge \overline{S_{p_s}}[\text{durability}] == \text{TransientLocal}) \\ \dashv \neg(\overline{S_{p_p}}[\text{liveliness}] == \text{Automatic} \wedge \overline{S_{p_s}}[\text{liveliness}] == \text{ManualByTopic}) \\ \dashv \neg(\overline{S_{p_p}}[\text{deadline}] > \overline{S_{p_s}}[\text{deadline}]) \quad \dashv \neg(\overline{S_{p_p}}[\text{duration}] > \overline{S_{p_s}}[\text{duration}]) \end{array}}{\Gamma \vdash \textbf{check\_qos}(x_1, x_2) : \texttt{bool}}$$

Figure 7: Auxiliary rule to check QoS settings.