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

PAULO CANELAS, Carnegie Mellon University, USA and University of Lisbon, Portugal
BRADLEY SCHMERL, Carnegie Mellon University, USA
ALCIDES FONSECA, University of Lisbon, Portugal
CHRISTOPHER S. TIMPERLEY, Carnegie Mellon University, USA

Component-based robot software frameworks, such as the Robot Operating System (ROS), allow developers to quickly compose and execute systems by focusing on configuring and integrating reusable, off-the-shelf components. However, these components often lack documentation on how to configure and integrate them correctly. Even when documentation exists, its natural language specifications are not enforced, resulting in misconfigurations that lead to unpredictable and potentially dangerous robot behaviors. In this work, we introduce ROSpec, a ROS-tailored domain-specific language designed to specify and verify component configurations and their integration. ROSpec's design is grounded in ROS domain concepts and informed by a prior empirical study on misconfigurations, allowing the language to provide a usable and expressive way of specifying and detecting misconfigurations. At a high level, ROSpec verifies the correctness of argument and component configurations, ensures the correct integration of components by checking their communication properties, and checks if configurations respect the assumptions and constraints of their deployment context. We demonstrate ROSpec's ability to specify and verify components by modeling a medium-sized warehouse robot with 19 components, and by manually analyzing, categorizing, and implementing partial specifications for components from a dataset of 182 misconfiguration questions extracted from a robotics Q&A platform.

CCS Concepts: • **Software and its engineering → Specification languages**; Software architectures; • **Computer systems organization → Robotics**.

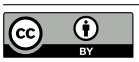
Additional Key Words and Phrases: Domain-Specific Language, Misconfigurations, Robot Operating System

ACM Reference Format:
Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. 2025. ROSpec: A Domain-Specific Language for ROS-Based Robot Software. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 391 (October 2025), 29 pages. https://doi.org/10.1145/3763169

1 Introduction

Component-based robot software allows developers to quickly compose and deploy their system by integrating reusable components [4]. Indeed, in recent years several frameworks, such as CARMEN [60], OROCOS [8], YARP [59], and the Robot Operating System (ROS) [69] 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 [74].

Authors' Contact Information: Paulo Canelas, pasantos@andrew.cmu.edu, pacsantos@ciencias.ulisboa.pt, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, USA and University of Lisbon, LASIGE, Lisboa, Portugal; Bradley Schmerl, schmerl@cmu.edu, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, USA; Alcides Fonseca, amfonseca@ciencias.ulisboa.pt, University of Lisbon, LASIGE, Lisboa, Portugal; Christopher S. Timperley, ctimperley@cmu.edu, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, USA.



ACM 2475-1421/2025/10-ART391
https://doi.org/10.1145/3763169

ROS is a framework built on top of Linux, which allows developers to focus on integrating and configuring reusable, off-the-shelf components [26, 46]. These *components* are processes that receive, process, and may produce information to other components. For example, the ROS ecosystem provides components for performing autonomous navigation, detecting obstacles and pedestrians, and translating motion sequences into motor commands [18, 19]. Developers often attempt to configure and integrate these components by searching for documentation and manually inspecting and understanding the components' source code.

However, these components often lack documentation on their usage, preventing their correct configuration [1, 26, 33]. Components are often configured through source code and several configuration files scattered around the system. Furthermore, as ROS-based systems are composed of dozens of different undocumented components with dependencies between each other, it becomes difficult to understand the overall architecture and configuration of the system.

Therefore, developers often have to rely on implicit and unverified assumptions, which may lead to misconfigurations [13]. *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]

As systems that interact with the physical world, ensuring correct component configuration and integration prior to execution is critical to preventing unpredictable and dangerous robot behaviors.

In this work, we introduce ROSpec, a ROS-tailored domain-specific language for specifying component configuration and integration. ROSpec abstracts away source-code implementation details and verifies correct component configuration and integration at the architectural level across specification files. Since ROS is the most widely adopted robotics framework, we use it as a proxy for broader component-based robot software development.

Domain-specific languages (DSL) have emerged as a promising approach for describing and specifying user intent about system architectures across various domains, including cloud computing [6], robotics [62], and cyber-physical systems [86]. Within the domain of robotics, DSLs have allowed developers to express and verify correct component interconnections [36, 61, 64, 89], real-time requirements [24, 30, 54, 57, 70], and hardware-software relationships [30, 78, 89]. However, the successful adoption of these languages in specific domains depends on several critical factors [43]. A language must be sufficiently expressive to allow developers to specify their architecture accurately [43], provide domain specialization to facilitate description [90], offer intuitive and comprehensible syntax [58], and maintain an appropriate level of abstraction that aligns with developers' understanding of the system [90]. However, to the best of our knowledge, no prior work has designed such a language that addresses these requirements to detect ROS-based misconfigurations.

We design ROSpec by using ROS-related concepts and studying misconfigurations identified in prior empirical work [13]. This allows the language to be expressive through the modeling of core domain concepts and the use of established programming language concepts, i.e., liquid and dependent types [73, 91]. In fact, ROSpec extends liquid types beyond their traditional application in executable languages like Haskell [84], LiquidJava [31], and Flux [49], by also allowing specifications over component configurations and the system's architecture by restricting their connections. 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

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

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 (Section 3);
- The introduction of ROSpec, a novel specification language that leverages domain-specific concepts and misconfiguration-related properties (Section 4);
- The formal definition of ROSpec's syntax and checking rules that use liquid types to restrict components configurations and their connection integration (Section 5);
- The evaluation of the language's expressivity by specifying a medium-sized robot case study (Section 6.1) and writing partial specifications of misconfigured components (Section 6.2).

2 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 [77]. 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 [19] supports autonomous navigation (e.g., planning, motion control, and localization); MoveIt! [18] performs motion planning and manipulation; and ROS Control [17] provides control algorithms and an interface for 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 [56]. 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,[3] one of the largest and most complex ROS autonomous systems, contains 230 components.

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

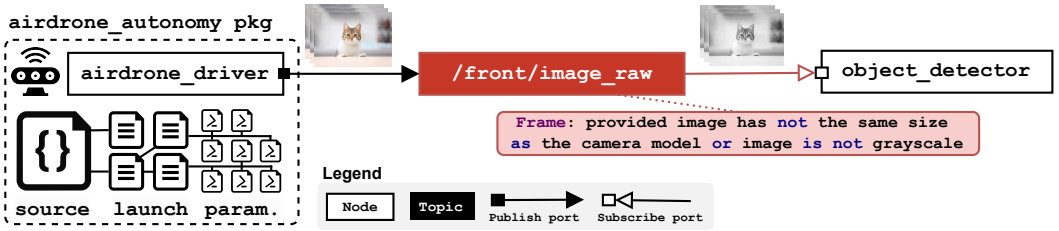


Fig. 1. Example of a misconfigured publisher-subscriber in ROS.[4] The `airdrone_driver` is composed of multiple source code, launch and parameter files. The subscriber expects grayscale images, but colored (RGB) images are provided. A possible solution is to introduce a node that converts colored images to grayscale.

Correctly configuring and integrating these components is critical, as configuration errors, or *misconfigurations* [2, 15, 63, 80], 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 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, 26, 81], 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.

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.

3.1 Methodology

Our methodology adapts a prior work's approach for language design that identifies properties from the analysis of developer errors [20]. We begin by collecting ROS domain knowledge and categories of misconfigurations. Then, we express the properties that model the collected knowledge. Figure 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.

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

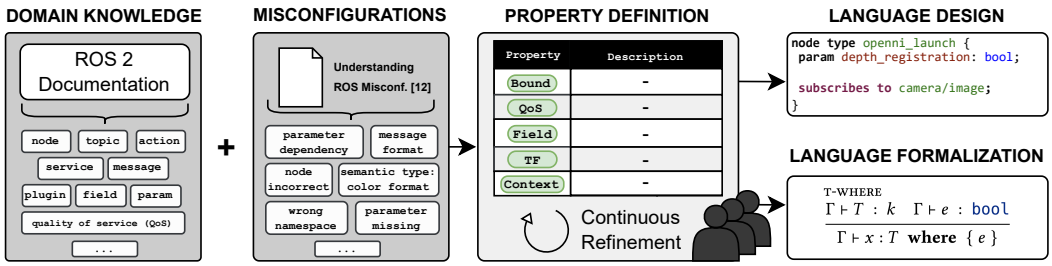


Fig. 2. Methodology for designing ROSpec based on ROS concepts and misconfiguration properties for ROS-based robotic systems. We begin by collecting ROS domain knowledge and information on domain knowledge and misconfigurations, using prior work in misconfigurations [13]. Then, we derive and iteratively refine properties from the collected knowledge. Finally, we design and formalize the specification language.

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 [56],[5] providing a glossary of domain concepts and their respective description.

The second source of knowledge comes from a prior empirical study on ROS misconfigurations [13]. 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 [13], 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 1 lists 12 core ROS domain concepts and their descriptions. We focus on the concepts most relevant to developers [77], 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` and `TF Broadcast/Listen` , and (iv) `Remapping` , which allows renaming and reusing components in their system.

Properties Definition. Table 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, allowing us to ensure 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

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

Table 1. Overview of the main domain-specific concepts in ROS 2, 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 during execution.
`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`	Change of nodes, arguments, parameters, and topics names at runtime without changing the code, 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.

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.

Table 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 values for arguments and parameters 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.
`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.

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 [79].

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[6] (now migrated to Robotics Stack Exchange),[7] 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`).

4.1 Node Definitions and Respective Parameter Refinements and Dependencies

```
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) -> elbow_joint/has_velocity_limits;
}
```

Listing (1) `Writer` specification of the `move_group` node, from MoveIt! [18], 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 (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`).[8]

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.

Listing 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

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

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 4.5)), 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 like `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 [31, 49, 73, 84]. 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 be `true`. This dependency is a documented requirement that velocity limits must be enabled when specifying velocity parameters.

Listing 2 presents an example of a parameter dependency misconfiguration from an example from ROS Answers [13]. In this case, the `Integrator` defined the **`system`**, which uses the specifications from all its components, such as Listing 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 1 as it requires velocity limits to be enabled when `max_velocity` is provided.

4.2 Messages Alias and Field Definition

ROSpec allows the `Writer` to encode domain knowledge about ROS into the types through type aliases. For instance, as presented in Listing 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) [14].

ROSpec also allows the definition of message aliases and their field specifications. For example, Listing 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.

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

```
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 (3) `Writer` specification of message and type aliases to provide documentation regarding image encodings and respective field information often missing from documentation (`Message`).[9]

4.3 Publisher & Subscriber, Service and Action Connections

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, 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 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). 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.

[10] https://answers.ros.org/question/164526

```
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) `Writer` specification of two nodes `hector_object_tracker` and `hector_map_server`, responsible for tracking objects in an environment and fusing perceptual data by publishing-subscribing data and providing-consuming services (`Publisher-Subscriber` , `Service & Action` , `Topic` , `Messages`).

```
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
+           get_distance_to_obstacle;
   }
}
```

Listing (5) `Integrator` configuration of both nodes from a ROS Answers question.[10] A misconfiguration is raised since no publisher is provided to `worldmodel/image_percept`, and due to a missing remap, no services are available for the `hector_object_tracker` node (`Connection` , `Messages`).

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.[12]

Listing 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 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

[11] https://answers.ros.org/question/142456
[12] 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 (6) `Writer` adapted specification of openni and custom node used in a ROS Answers question. Components connections contain two policies (`Publisher-Subscriber` , `Quality of Service`).[11]

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

Listing (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 incorrect component integration. (`Connection` , `QoS`).

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.

4.5 TF Transforms and Contextual Information

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 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

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

```
node type laser_scan_matcher_type {
  context is_simulation: bool;
  // multiple parameters here...
  optional param use_sim_time: bool = false;

  broadcast world to base_link;
  listens base_link to laser;
} where {
  is_simulation -> use_sim_time;
}
```

Listing (8) `Writer` specification of `laser_scan_matcher` node containing frame broadcast and listens, and execution context information (`TF Frames` , `TF Broadcast/Listen`).

```
system {
  node instance laser_scan_matcher:
                laser_scan_matcher_type {
      context is_simulation = true;
  }
}
```

Listing (9) `Integrator` configuration leading to context and broadcast errors (`Context` , `TF-Listen` , `TF-Graph`).[13]

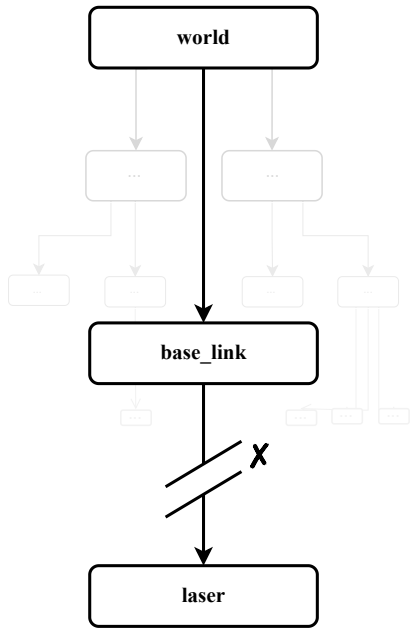


Fig. 7. Faulty branch of TF tree due to missing broadcast from `base_link` to `laser`.

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 with simulated time instead of system time.

In Listing 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 7 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.

4.6 Plugins

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 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.

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

```
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 (10) `Writer` perspective of a node and plugin definition from a ROS Answers question (`Plugin`).[14]

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.

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.

5.1 Grammar

ROSpec is designed based on the concepts identified in Table 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 [13]. The grammar of ROSpec is defined in Figure 9.

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 [73], 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 6 that these can be expressive enough to detect several real-world misconfigurations.

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

Fig. 9. A subset of the grammar for Declarations and top-level definitions, as well as types and typing context.

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.

5.2 Formation Rules

The typing rules in Figure 10 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

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

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

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

Fig. 10. Definition Formation Rules, $\boxed{\Gamma \vdash D \dashv \Gamma}$.

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 specification;[15] 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.

$$\textsc{S-Int}\quad \frac{}{\Gamma \vdash \texttt{int} <: \texttt{int}} \qquad \textsc{S-Bool}\quad \frac{}{\Gamma \vdash \texttt{bool} <: \texttt{bool}} \qquad \textsc{S-Struct}\quad \frac{\overline{\Gamma \vdash T_i <: U_i}}{\Gamma \vdash \textbf{struct}\{\ \overline{x_i : T_i}\ \} <: \textbf{struct}\{\ \overline{x_i : U_i}\ \}}$$

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

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

Fig. 11. Subtyping Rules, $\boxed{\Gamma \vdash t_1 <: t_2}$.

Figure 11 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

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

a concrete value into a refined type that only describes itself [65]. 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 [22], (e.g., $x > 1 \implies x > 0$) which only occurs in message types [73].

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.

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

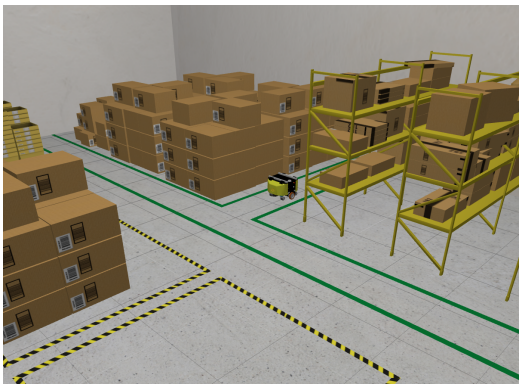


Fig. 12. AWS Small Warehouse World.

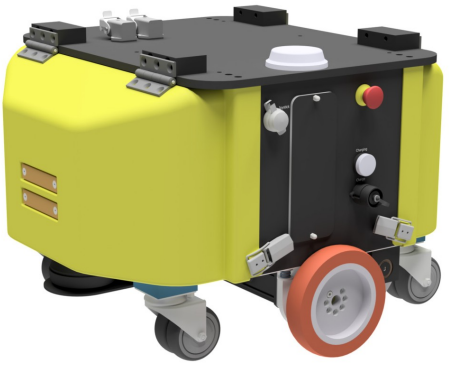


Fig. 13. Neobotix MP-400.

System Details. Our case study was taken from a popular online robotics learning platform, The Construct Sim,[17], 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 [15], we expect that providing ROSpec specifications to platforms like The Construct Sim can help newcomers detect and understand misconfigurations while learning core ROS concepts.

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 12) where robots navigate and transport loads [68]. The Neobotix MP-400 (Figure 13) 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.

[16] https://docs.nav2.org/configuration/packages/configuring-amcl.html
[17] 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 (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.[16] The complete case study code is provided in the artifact.

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 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.

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 [13]. 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

Table 3. Summary of results from manual analysis and specification of 182 questions from prior work [13]. 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%)

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 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 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 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.[18]

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 [77], 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 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.

7 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 [62, 86]. 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 [36, 61, 64, 89], ensuring that software components are correctly assembled. Behaviorally, these languages specify properties to ensure real-time (e.g., frequency [30], queues [5], and

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

synchronization [24]), timing [54, 70], and other functional [36, 57] and non-functional [55, 71] requirements. Domain architectural languages also specify hardware properties [30, 78, 89] to ensure the system's software-to-hardware architectural consistency. Finally, synchronous programming languages such as Lustre [38], Esterel [7], and coordination languages like Lingua Franca [53] and Kahn Process Networks [35] 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 [3], Architecture Analysis and Design Language (AADL) [27], and Acme [34], have previously been used to describe robot architectures [10, 75]. Specifically to ROS, recent work has described component connections using AADL [5]. However, its application in other domains identified usability challenges, including property ambiguity regarding system requirements and subcomponent specifications (which ROSpec addresses directly) [23], increased entry barriers for adoption [23], and lack of flexibility in supporting user-defined connectors [66]. When specifically comparing AADL to DSLs, such as ROSpec, prior work identifies three main challenges [11]: (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 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 [76] and ROSDiscover [79], recover the structural architecture of the system and can detect common connection issues (e.g., subscriber missing a publisher), ROSInfer [25] infers reactive, periodic, and state-dependent information from components, allowing them to detect three behavioral architectural misconfigurations, and Phys [44] 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.

Liquid types [73] have traditionally been used in executable languages, such as Haskell [84], TypeScript [85], Java [31], and Flux [49] to ensure program correctness, verify resource consumption [45], and synthesize programs given a specification [29, 67, 82, 88]. While recent work has explored applying refinement types to non-executable languages for visualization synthesis [92] and security in web applications [50], 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.

8 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 [72] 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 [21]. Microservices architectures could benefit from extending JSON-based descriptions like the Microservices Architecture Language (MIRL) [51] with liquid types to refine service nodes and their relationships. Similarly, IoT and edge computing architectures [39] could use liquid and dependent types for formal specifications of resource constraints (e.g., memory, bandwidth, and latency) and security requirements [47, 48]. 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[19] 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.

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

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 [90]. When designing ROSpec, we incorporated ROS 2 concepts to provide familiarity with concepts developers use, while drawing insights on prior research [13] 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 [83]. 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 [32]. These provide the understanding of how they specify their components, identify their challenges, and understand the features needed for specification and misconfiguration detection.

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 [76], ROSDiscover [79], and ROSInfer [25] can collect configurations and identify component connections. Recent advances in misconfiguration detection — including test-based approaches [16], static and dynamic analysis [40, 41, 93], and Large Language Models [52] — could complement these architectural tools to identify misconfigurations at the source-code level.

However, as discussed in Section 7, 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 [25, 76, 79]. Concurrently, LLM-based techniques have shown promise in converting natural language specifications into formal specifications [37, 87]. 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 [9, 28, 42] that detect misconfigurations against properties defined in the message field types.

9 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.

10 Data-Availability Statement

The complete dataset of 182 ROS Answers questions with categories and specifications, the ROSpec source-code and its language design, including the glossary of used core ROS concepts, grammar definitions and type-checking rules, the warehouse robot case study source-code and specification, and a website with language documentation, are available at 10.5281/zenodo.15722060 [12].

Acknowledgments

We thank the reviewers for their feedback that improved our work. Additionally, we would like to thank Claire Le Goues, Jonathan Aldrich, and Catarina Gamboa for their feedback on this work. 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.

References

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. A Study on Challenges of Testing Robotic Systems. In *International Conference on Software Testing*. 96–107. https://doi.org/10.1109/ICST46399.2020.00020

[2] Nicholas Albergo, Vivek Rathi, and John-Paul Ore. 2022. Understanding Xacro Misunderstandings. In *International Conference on Robotics and Automation*. 6247–6252. https://doi.org/10.1109/ICRA46639.2022.9812349

[3] Robert Allen. 1997. *A Formal Approach to Software Architecture*. Ph. D. Dissertation. Carnegie Mellon Univeristy School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

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

[5] Gianluca Bardaro, Andrea Semprebon, Agnese Chiatti, and Matteo Matteucci. 2019. From Models to Software Through Automatic Transformations: An AADL to ROS End-to-End Toolchain. In *International Conference on Robotic Computing*. 580–585. https://doi.org/10.1109/IRC.2019.00118

[6] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *Comput. Surveys* (2018), 22:1–22:38. https://doi.org/10.1145/3150227

[7] Gérard Berry. 2000. The foundations of Esterel. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). 425–454.

[8] Herman Bruyninckx. 2001. Open Robot Control Software: the OROCOS project. In *International Conference on Robotics and Automation*. 2523–2528. https://doi.org/10.1109/ROBOT.2001.933002

[9] Ricardo Caldas, Juan Antonio Piñera García, Matei Schiopu, Patrizio Pelliccione, Genaína Rodrigues, and Thorsten Berger. 2024. Runtime Verification and Field-Based Testing for ROS-Based Robotic Systems. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2544–2567. https://doi.org/10.1109/TSE.2024.3444697

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

[11] Paulo Canelas, Bradley Schmerl, Alcides Fonesca, and Christopher S. Timperley. 2025. The Usability Argument for ROS-based Robot Architectural Description Languages. In *Annual Workshop On The Intersection of HCI and PL*. https://doi.org/10.1184/R1/29087072.v1

[12] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher Timperley. 2025. *Artifact for "rospec: A Domain-Specific Language for ROS-based Robot Software"*. https://doi.org/10.5281/zenodo.15722060

[13] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. 2024. Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches. In *International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3650212.3680350

[14] Paulo Canelas, Trenton Tabor, John-Paul Ore, Alcides Fonseca, Claire Le Goues, and Christopher Steven Timperley. 2024. Is it a Bug? Understanding Physical Unit Mismatches in Robot Software. In *International Conference on Robotics and Automation*. 1–7. https://doi.org/10.1109/ICRA57147.2024.10611413

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

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

[17] 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. 2017. ros_control: A generic and simple control framework for ROS. *The Journal of Open Source Software* 2 (2017), 456. https://doi.org/10.21105/JOSS.00456

[18] Sachin Chitta, Ioan Alexandru Sucan, and Steve Cousins. 2012. MoveIt! [ROS Topics]. *Robotics & Automation Magazine* 19, 1 (2012), 18–19.

[19] David Coleman, Ioan A. Șucan, Sachin Chitta, and Nikolaus Correll. 2014. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. *Journal of Software Engineering for Robotics* (2014), 3–16. http://arxiv.org/abs/1404.3785

[20] 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. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 670–697. https://doi.org/10.1145/3649835

[21] João Pedro Vieira David. 2021. *Improving Machine Learning Pipeline Creation using Visual Programming and Static Analysis*. Master's thesis. Universidade de Lisboa.

[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. 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*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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

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

[25] Tobias Dürschmid, Christopher Steven Timperley, David Garlan, and Claire Le Goues. 2024. ROSInfer: Statically Inferring Behavioral Component Models for ROS-based Robotics Systems. In *International Conference on Software Engineering*. 144:1–144:13. https://doi.org/10.1145/3597503.3639206

[26] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. 2019. The Robot Operating System: Package reuse and community dynamics. *Journal of Systems and Software* (2019), 226–242.

[27] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley.

[28] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. 2020. ROSMonitoring: A Runtime Verification Framework for ROS. Springer International Publishing, 387–399.

[29] Alcides Fonseca, Paulo Santos, and Sara Silva. 2020. The Usability Argument for Refinement Typed Genetic Programming. In *International Conference on Parallel Problem Solving from Nature*. Springer, 18–32. https://doi.org/10.1007/978-

3-030-58115-2_2

[30] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. 2013. A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms. arXiv:1301.7190 [cs.RO]

[31] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *International Conference on Software Engineering*. 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132

[32] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. In *Conference on Programming Language Design and Implementation*. 26 pages. https://doi.org/10.1145/3729327

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

[34] David Garlan, Robert T. Monroe, and David Wile. 2000. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman (Eds.). Cambridge University Press, 47–68.

[35] Marc Geilen and Twan Basten. 2003. Requirements on the Execution of Kahn Process Networks. In *Programming Languages and Systems, European Symposium on Programming*, Pierpaolo Degano (Ed.), Vol. 2618. 319–334. https://doi.org/10.1007/3-540-36575-3_22

[36] Nicolas Gobillot, Charles Lesire, and David Doose. 2014. A Modeling Framework for Software Architecture Specification and Validation. In *Simulation, Modeling, and Programming for Autonomous Robots*. 303–314. https://doi.org/10.1007/978-3-319-11900-7_26

[37] Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal Specifications from Natural Language. arXiv:2206.01962 [cs.SE] https://arxiv.org/abs/2206.01962

[38] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 2002. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (2002), 1305–1320.

[39] Najmul Hassan, Saira Andleeb Gillani, Ejaz Ahmed, Ibrar Yaqoob, and Muhammad Imran. 2018. The Role of Edge Computing in Internet of Things. *IEEE Communications Magazine* 56, 11 (2018), 110–115. https://doi.org/10.1109/MCOM.2018.1700906

[40] Md. Abir Hossen, Sonam Kharade, Jason M. O'Kane, Bradley R. Schmerl, David Garlan, and Pooyan Jamshidi. 2024. CURE: Simulation-Augmented Auto-Tuning in Robotics. *CoRR* abs/2402.05399 (2024). https://doi.org/10.48550/ARXIV.2402.05399

[41] 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. 2023. CaRE: Finding Root Causes of Configuration Issues in Highly-Configurable Robots. *IEEE Robotics Automation Letters* 8, 7 (2023), 4115–4122. https://doi.org/10.1109/LRA.2023.3280810

[42] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. 2014. ROSRV: Runtime Verification for Robots. In *Runtime Verification*. 247–254.

[43] Sajjad Hussain. 2013. *Investigating Architecture Description Languages (ADLs) A Systematic Literature Review*. Master's thesis. Linköping University, Software and Systems, The Institute of Technology.

[44] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian G. Elbaum, and Zhaogui Xu. 2018. Phys: probabilistic physical unit assignment and inconsistency detection. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 563–573.

[45] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid resource types. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 106:1–106:29. https://doi.org/10.1145/3408988

[46] Sophia Kolak, Afsoon Afzal, Michael Hilton, Claire Le Goues, and Christopher S Timperley. 2020. It Takes a Village To Build a Robot: An Empirical Study of the ROS Ecosystem. In *International Conference on Software Maintenance and Evolution*. 430–440.

[47] David Kolevski and Katina Michael. 2024. Edge Computing and IoT Data Breaches: Security, Privacy, Trust, and Regulation. *IEEE Technology and Society Magazine* 43, 1 (2024), 22–32. https://doi.org/10.1109/MTS.2024.3372605

[48] Anit Kumar and Dhanpratap Singh. 2025. Securing IoT devices in edge computing through reinforcement learning. *Computers and Security* 155 (2025), 104474. https://doi.org/10.1016/J.COSE.2025.104474

[49] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proceedings of the ACM Programming Languages* 7, Conference on Programming Language Design and Implementation (2023), 1533–1557. https://doi.org/10.1145/3591283

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

[51] Luka Lelovic, Michael Mathews, Amr S. Abdelfattah, and Tomás Cerný. 2023. Microservices Architecture Language for Describing Service View. In *International Conference on Cloud Computing and Services Science*. 220–227. https:

//doi.org/10.5220/0011850200003488
[52] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2024. Large Language Models as Configuration Validators. In *International Conference on Software Engineering*. 204–216.
[53] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems* 20, 4 (2021), 36:1–36:27. https://doi.org/10.1145/3448128
[54] Markus Look, Antonio Navarro Perez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2014. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. arXiv:1409.2388 [cs.SE]
[55] Markus Luckey and Gregor Engels. 2013. High-quality specification of self-adaptive software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 143–152. https://doi.org/10.1109/SEAMS.2013.6595501
[56] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* (2022). https://doi.org/10.1126/scirobotics.abm6074
[57] Jabier Martinez, Alejandra Ruiz, Ansgar Radermacher, and Stefano Tonetta. 2021. Assumptions and Guarantees for Composable Models in Papyrus for Robotics. In *International Workshop on Robotics Software Engineering*. 1–4. https://doi.org/10.1109/RoSE52553.2021.00007
[58] Nenad Medvidovic and Richard N. Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *Transactions of Software Engineering* 1 (2000), 70–93. https://doi.org/10.1109/32.825767
[59] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. 2006. YARP: Yet Another Robot Platform. *International Journal of Advanced Robotic Systems* 3, 1 (2006), 8. https://doi.org/10.5772/5761
[60] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. 2003. Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit. In *International Conference on Intelligent Robots and Systems*. IEEE, 2436–2441. https://doi.org/10.1109/IROS.2003.1249235
[61] Henry Muccini and Mohammad Sharaf. 2017. CAPS: Architecture Description of Situational Aware Cyber Physical Systems. In *International Conference on Software Architecture*. 211–220. https://doi.org/10.1109/ICSA.2017.21
[62] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. 2016. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering in Robotics* (2016), 75–99.
[63] John-Paul Ore, Sebastian G. Elbaum, and Carrick Detweiler. 2017. Dimensional inconsistencies in code and ROS messages: A study of 5.9M lines of code. In *International Conference on Intelligent Robots and Systems*. 712–718.
[64] Francisco J. Ortiz, Diego Alonso, Francisca Rosique, Francisco Sánchez-Ledesma, and Juan Angel Pastor. 2014. A Component-Based Meta-Model and Framework in the Model Driven Toolchain C-Forge. In *Simulation, Modeling, and Programming for Autonomous Robots*. 340–351. https://doi.org/10.1007/978-3-319-11900-7_29
[65] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics, International Conference on Theoretical Computer Science*. 437–450. https://doi.org/10.1007/1-4020-8141-3_34
[66] Mert Ozkaya and Christos Kloukinas. 2013. Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. In *Euromicro Conference on Software Engineering and Advanced Applications*. 177–184. https://doi.org/10.1109/SEAA.2013.34
[67] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Conference on Programming Language Design and Implementation*. 522–538. https://doi.org/10.1145/2908080.2908093
[68] Proteus. [n. d.]. *I'm Amazon's first autonomous robot. Follow me around on my typical workday (watercooler break included).* https://www.aboutamazon.com/news/operations/amazon-robotics-autonomous-robot-proteus-warehouse-packages
[69] Morgan Quigley, Ken Conle, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. 2009. ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software* (2009), 1–6.
[70] Akshay Rajhans, Shang-Wen Cheng, Bradley Schmerl, David Garlan, Bruce Krogh, Clarence Agbi, and Ajinkya Bhave. 2009. An Architectural Approach to the Design and Analysis of Cyber-Physical Systems. *Electronic Communication of the European Association of Software Science and Technology* (2009). https://doi.org/10.14279/tuj.eceasst.21.286
[71] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. 2017. An Extensible Model-Based Framework for Robotics Software Development. In *International Conference on Robotic Computing*. 73–76. https://doi.org/10.1109/IRC.2017.21
[72] Nicholas Rescher and Alasdair Urquhart. 2012. *Temporal logic*. Vol. 3. Springer Science & Business Media.
[73] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Conference on Programming Language Design and Implementation*. ACM, 159–169. https://doi.org/10.1145/1375581.1375602
[74] ROS-Industrial. 2021. Current members - ROS-industrial. https://rosindustrial.org/ric/current-members/

[75] Ivan Ruchkin, Bradley R. Schmerl, and David Garlan. 2015. Architectural Abstractions for Hybrid Programs. In *Symposium on Component-Based Software Engineering*. 65–74. https://doi.org/10.1145/2737166.2737167
[76] André Santos, Alcino Cunha, and Nuno Macedo. 2021. The High-Assurance ROS Framework. In *International Workshop on Robotics Software Engineering*. 37–40. https://doi.org/10.1109/ROSE52553.2021.00013
[77] André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. 2017. Mining the usage patterns of ROS primitives. In *International Conference on Intelligent Robots and Systems*. 3855–3860. https://doi.org/10.1109/IROS.2017.8206237
[78] Amita Singh, Fabian Quint, Patrick Bertram, and Martin Ruskowski. 2019. A Framework for Semantic Description and Interoperability across Cyber-Physical Systems. *International Journal on Advances in Intelligent Systems* (2019), 70–81.
[79] Christopher Steven Timperley, Tobias Dürschmid, Bradley R. Schmerl, David Garlan, and Claire Le Goues. 2022. ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems. In *International Conference on Software Architecture*. 112–123. https://doi.org/10.1109/ICSA53651.2022.00019
[80] Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wąsowski. 2024. ROBUST: 221 bugs in the Robot Operating System. *Empirical Software Engineering* 29, 3 (2024), 57.
[81] Daniella Tola and Peter Corke. 2023. Understanding URDF: A Survey Based on User Experience. In *International Conference on Automation Science and Engineering*. 1–7. https://doi.org/10.1109/CASE56687.2023.10260660
[82] Sabrina Tseng, Erik Hemberg, and Una-May O'Reilly. 2022. Synthesizing Programs from Program Pieces Using Genetic Programming and Refinement Type Checking. In *European Conference on Genetic Programming*. 197–211. https://doi.org/10.1007/978-3-031-02056-8_13
[83] Aaron Turon. [n. d.]. *Rust's language ergonomics initiative*. https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html
[84] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *International Conference on Functional Programming*. ACM, 269–282. https://doi.org/10.1145/2628136.2628161
[85] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Conference on Programming Language Design and Implementation*. ACM, 310–325. https://doi.org/10.1145/2908080.2908110
[86] Kaiyu Wan, Ka Lok Man, and Danny Hughes. 2010. Specification, Analyzing Challenges and Approaches for Cyber-Physical Systems (CPS). *Engineering Letters* (2010).
[87] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 36. 65030–65055. https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf
[88] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151
[89] Hongxing Wei, Xinming Duan, Shiyi Li, Guofeng Tong, and Tianmiao Wang. 2009. A component based design framework for robot software architecture. In *International Conference on Intelligent Robots and Systems*. 3429–3434. https://doi.org/10.1109/IROS.2009.5354161
[90] Eoin Woods and Rich Hilliard. 2005. Architecture Description Languages in Practice Session Report. In *Working Conference on Software Architecture*. 243–246. https://doi.org/10.1109/WICSA.2005.15
[91] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Symposium on Principles of Programming Languages*, Andrew W. Appel and Alex Aiken (Eds.). 214–227. https://doi.org/10.1145/292540.292560
[92] Jingtao Xia, Junrui Liu, Nicholas Brown, Yanju Chen, and Yu Feng. 2024. Refinement Types for Visualization. In *International Conference on Automated Software Engineering*. ACM, 1871–1881. https://doi.org/10.1145/3691620.3695550
[93] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485517

Received 2025-03-25; accepted 2025-08-12