An Experience Report on Challenges in Learning the Robot Operating System

Paulo Canelas*
Miguel Tavares
Ricardo Cordeiro
Alcides Fonseca
{pacsantos,alcides}@ciencias.ulisboa.pt
{mtavares,rcordeiro}@lasige.di.fc.ul.pt
LASIGE, Departamento de Informática,
Faculdade de Ciências da Universidade de Lisboa
Portugal

Christopher S. Timperley
ctimperley@cmu.edu
Institute for Software Research,
School of Computer Science,
Carnegie Mellon University
USA

ABSTRACT

The Robot Operating System (ROS) was initially introduced to lower the barriers to robots software development by reducing the need for extensive domain knowledge. ROS allows developers to build valuable robots by configuring and reusing off-the-shelf components while writing little, if any, code through its modular design, loosely coupled architecture, and rich package ecosystem. However, despite the advantages of this approach, the lack of documentation can present a challenge to novice users.

In this work, we discuss the challenges and experience of learning and using ROS from the perspective of three novice users with little to no prior experience in robotics. We report on the experiences in learning ROS through a popular commercial training course provided by The Construct Sim. Through our analysis, we identify several common misunderstandings, mistakes, and bugs, and we outline possible improvements to ROS to overcome these challenges.

Our findings motivate further studies on the development of robotic systems in ROS by novice users and promote the improvement of the ROS ecosystem, on educational and training materials of ROS, and on tooling development to help novices identify and correct simple mistakes.

CCS CONCEPTS

• **Software and its engineering** → *Software usability*; • **Computer systems organization** → **Robotics**.

KEYWORDS

Robot Operating System, Python, Developer Experience, Usability

*Also with the Institute for Software Research and School of Computer Science of Carnegie Mellon University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RoSE'22, May 9, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9317-1/22/05…$15.00
https://doi.org/10.1145/3526071.3527521

ACM Reference Format:
Paulo Canelas, Miguel Tavares, Ricardo Cordeiro, Alcides Fonseca, and Christopher S. Timperley. 2022. An Experience Report on Challenges in Learning the Robot Operating System. In *4th International Workshop on Robotics Software Engineering (RoSE'22), May 9, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3526071.3527521

1 INTRODUCTION

Programming a robot encompasses many challenges that range from integrating various hardware components to writing and composing the high-level programming logic.

The Robot Operating System (ROS) [6], colloquially known as the "Linux of Robotics" [10], is one of the most popular frameworks for this purpose that tackles these challenges through modularity and reusability.

In ROS, developers can reuse existing components in their robots, thus abstracting the implementation details of several components of their robot, from odometry to route planning.

On the one hand, abstracting away the algorithmic and implementation details of common components (e.g., path planning, object detection) eliminates the need for advanced domain expertise and opens up robotics software development to a far wider audience. On the other hand, the mistakes of novice programmers may result in costly and possibly dangerous outcomes due to the robot's interaction with its physical environment [3]. Interventions that aid developers in identifying and avoiding such mistakes are therefore highly desirable (e.g., domain-specific feedback in programming environments).

In this work, we aim to identify the challenges that novices face when programming robots. By identifying the most frequent and time-consuming challenges, we can guide the research and development of approaches to improve the usability of ROS, such as static analysis tools, better visualization, documentation, and training materials. We focus specifically on ROS as it is one of the most popular frameworks and is designed to reduce complexity.

To identify the challenges, three investigators (the first three authors), all of whom had no prior ROS experience, took a widely available, introductory, paid course on ROS provided by The Construct Sim.[1] Throughout the course, the investigators catalogued their experiences, focusing on the challenges that they faced and

[1] https://app.theconstructsim.com

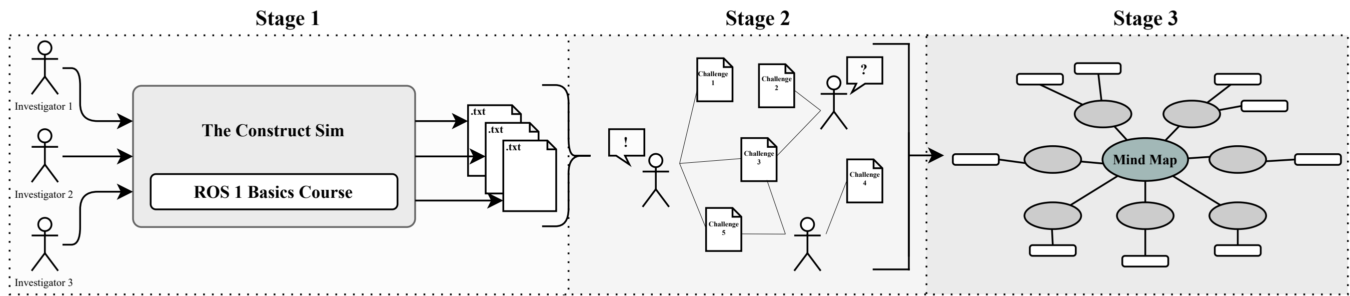

Figure 1: Methodology of the challenges and problems annotation and categorization divided in three stages. Stage 1: the investigators are exposed to the ROS Basics in 5 Days (Python) course and each one annotates the challenges and problems encountered during the learning. Stage 2: the unorganized notes are categorized and the investigators discuss the shared challenges. Stage 3: the creation of a mind map from the organized categories of challenges and problems identified by the investigators.

the extent to which those challenges hindered their progress. Afterward, the three investigators systematically discussed, compared, and categorized the challenges that they experienced. In this paper, we report those shared experiences. While this work is based on the experience of only three people, we provide valuable insights and a thorough description of each problem to the extent that may not be possible with a larger cohort (e.g., via questionnaires or interviews as in recent studies of challenges in robotics software engineering. [1, 2]).

2 METHODOLOGY

Figure 1 illustrates our study methodology: First, each of the three investigators took the ROS Basics in 5 Days (Python) course at The Construct Sim. During the course, each investigator maintained notes on any difficulties encountered. Upon completing the course, the investigators met to discuss and compare their experiences before consolidating their challenges into a mind map consisting of seven top-level categories, each of which consisted of one or more sub-categories describing specific issues.

In the rest of this section, we present the threats to validity of this work (Section 2.1), we describe the background of each of the investigators (Section 2.2), and the course that was taken (Section 2.3).

2.1 Threats to Validity

We identify the following threats to validity in this work:

- This work is based on the investigation of only three users. We accept this limitation and encourage others to replicate this experience. However, we provide a detailed report that might not be possible if the target included many users. It is also important to note the difficulty in recruiting participants, as this task requires weeks of effort and participants need to be interested in learning ROS, but have not learned anything about it yet. This experience can also help guiding the design of questionnaires for a larger audience.
- Our conclusions are biased by the course followed by the investigators, and certain challenges might be associated with the course and not necessarily in ROS. To mitigate this issue, we selected the best resource available for learning ROS. If there is no better alternative, maybe the limitations of this particular course reflect the lack of better materials for this purpose.
- Our conclusions might be too specific for ROS1, while ROS2 is expected to be the standard in the near future. ROS2 improves the architecture design over ROS1, but the new, more modular approach also increases the indirection between the high-level code and the behaviors of modules. Thus, we find that most of the problems reported can also apply to ROS2.

2.2 Investigators

This study reports the experiences of three investigators, corresponding to the three first authors of the paper: **Investigator 1** (Tavares) and **Investigator 2** (Cordeiro) were final-year M.Sc. students in Computer Science at University of Lisbon, and **Investigator 3** (Canelas) was a second-year Ph.D. student in Software Engineering at the same institution and at Carnegie Mellon University. All three investigators shared a similar curricular background and had prior experience with Java, C, and Python. Only one investigator (Tavares) had prior experience of programming robots using Thymio [7], while the other investigators had no such experience.

2.3 The Construct Sim

The Construct Sim is an online learning platform for ROS and robotics that provides over 40 courses covering everything from the basic concepts of ROS to programming autonomous vehicles.

This paper reports on the developer experience of the investigators in an introductory course on ROS. The investigators took the Python version of *ROS Basics in 5 Days* instead of the C++ version, as they were familiar in the language and learning another programming language would introduce another variable in the study.

The *ROS Basics in 5 Days (Python)* is split into four parts: an introduction to ROS; a section on the publisher and subscriber model; another on the services client and services server; and a final section on action clients and servers. Each part provides code examples that follow ROS best practices [4]. The course allows its users to interact with four different simulations of robotic systems: a BB8

Table 1: Tasks exercised by the investigators during the learning in the ROS Basics in 5 Days (Python) course.

Module Name	Task ID	Task Description	Simulation
Publisher	T1	Publish data to the cmd_vel topic to induce movement in the robot.	Turtlebot
Subscriber	T2	Create a subscriber that prints the robot odometry.	
	T3	Create a Publisher that publishes the robots age.	
	T4	Create a new message type called Age.msg.	
Publisher-Subscriber	T5	Quiz: Create a publisher that moves the robot. Create a subscriber that reads information from the scanner. The robot should avoid a wall as it gets closer.	
Services Clients	T6	Create a launch file which launchs a service.	Wam Arm
	T7	Get information on the service messages a service uses.	
	T8	Create a client that requests the arm to follow a trajectory.	
Services Servers & Messages	T9	Create a service which moves the robot in a circle.	BB8 Robot
	T10	Create a client to call the new service and induce movement in the robot.	
	T11	Create a custom service message with the duration the robot should move.	
	T12	Adapt the service server to receive and process the new type of message.	
Services	T13	Quiz: Create a service that moves the robot in a square. The side of the square and number of repetitions is given by a new type of message. Create a service client and a launch file.	
Actions Clients	T14	Use the command line to takeoff, move and land the drone.	Drone
	T15	Experiment the commands to obtain information about an action.	
	T16	Observe the differences between the synchronous and asynchronous clients.	
	T17	Create a package which launches an action client.	
	T18	Have the drone move and take pictures during flight.	
	T19	Practice in the command line the different interactions with the actions.	
Actions Servers	T20	Create an action server that moves the drone in a square.	
Actions	T21	Quiz: Create an action server whichs receives TAKEOFF or LAND as a goal. As feedback, it publishes the current action taking place.	
Real Robot Lab	T22	Create a system that makes the robot move in a square. The robot should stop if an object is detected infront. During its execution, the robot publishes the total amount of meters travelled. As result it should record one measurement per second of the robots position.	Turtlebot

robot, a Wam robot-arm, a TurtleBot 3, and a drone. The investigators completed the course using an online web version of VSCode featuring an integrated simulator, Gazebo [5].

The beginning of the course provided a brief introduction to the Robot Operating System and information on building custom ROS projects. Throughout the course, the investigators were required to accomplish different tasks (described in Table 1), to apply the primary concepts taught in each module.

Each module contains a final quiz that evaluates the investigators learning on that specific topic. After completing the quiz for each module, the investigators were required to do a final course that combines the three ROS architectural models. The final exam allowed the execution of the robotic system both in simulation and with a real robot.

3 RESULTS

Figure 2 provides a breakdown of the challenges of using ROS that the investigators identified. Below, we discuss each challenge in more detail and use shaded circles to indicate the number of investigators that faced each particular challenge. We also identify in which tasks each challenged occurred, using the identifiers from Table 1.

3.1 Build System

3.1.1 IDL File Consistency (●●○).

The first challenge faced by the investigators was the complexity in defining new message formats. Because this process requires changing code in multiple places in different files [**T11**, **T13**], it was always necessary to follow the tutorial, even after repeating this process multiple times. Furthermore, the investigators also made the common mistake of forgetting some dependencies when creating a package. Although these are issues that occur in other programming domains, IDE and build systems often diagnose and automate the process of propagating these changes. This issue might also be caused by an highly coupled design regarding message types.

A second challenge when operating with the IDL is keeping the dependencies between the launch files and the Python implementation files consistent [**T5**]. When creating a new ROS node, developers provide a unique identifier that can be used to start that node from a launch file. However, the system does not have a sanity check between those two identifiers. In the case of a mismatch (common with typographical error), the robot simulation

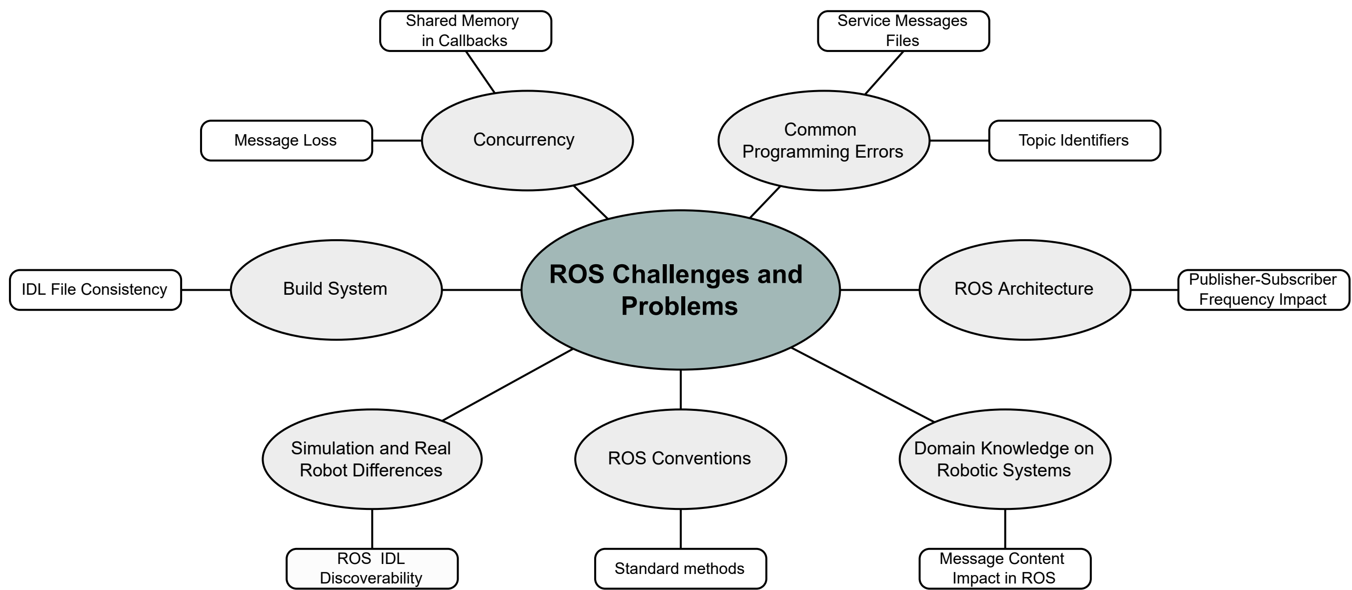

Figure 2: Mind map of the challenges and problems identified by the investigators.

does not have the proper behavior and identifying this mismatch as the source of the program is not straightforward.

3.2 ROS Interface Description Language

The ROS Interface Description Language (IDL) is used to describe the format of messages, services, and actions. ROS transforms IDL descriptions into concrete implementations of those formats for different target languages (e.g., C++ and Python).

3.2.1 ROS IDL Discoverability (●●●).
ROS provides components for different common tasks in robots. However, identifying which component is responsible for providing certain information or functionality was challenging.

For instance, the investigators used TurtleBot 3 and a drone over two simulations. When asked to obtain information about the position of the robot, they struggled to identify the topic in which this information was published for the drone [**T20**]. While the standard topic for obtaining a robots position and orientation is the **`odom`** topic, the drone uses **`gt_pose`**. Although for experienced users, it may be evident why odometry is only applied for wheeled mobile robots, this is not true for novice users with no background experience in robotics. The investigators were required to blindly search each topic until they found the right one. Furthermore, it is not explicit how each message and its parameters impact the execution of the robotic system due to a lack of documentation.

3.3 ROS Conventions

3.3.1 Standard methods (●●○).
Due to their lack of experience, the investigators did not follow expected good practices in ROS. One example is forgetting to implement callbacks and hook methods (e.g., `on_shutdown` for a safe exit) [**T1**, **T5**, **T21**], typically required for the good functioning of the robotic system. While this has been introduced in the course, this was frequently missed in the process of implementing new features, with no clear message identifying this issue.

3.4 ROS Architecture

3.4.1 Publisher-Subscriber Frequency Impact (●●●).
Different ROS components require different event frequencies. For example, a component may need to perform an action each millisecond, but the component that provides the required information only emits updated information each second. ROS developers can use both components in a project, without understanding that there is a mismatch in the assumed and provided information frequencies.

To better illustrate, consider Listing 1, which presents a simple example of a node that publishes continuously publishes messages to the **`cmd_vel`** topic to move the robot in the x axis. The overall code can be split into three parts: 1) the creation of the node and publisher; 2) the creation of the message being published; and 3) the constant publish of the message to move the robot. Although this presents a very naive example of a publisher, during the course we identified three associated challenges.

The first challenge arises in the definition of the publisher. One is required to define the `queue_size` to the publisher. Both ROS publishers and subscribers place their messages on a bounded queue to await processing. The investigators found it difficult to determine queue sizes and predict their impact, and instead relied on choosing arbitrary values and testing them on the robot [**T1**, **22**]. The definition of a good queue size is typically associated with the frequency at which the message is being published.

Defining the right frequency corresponds to the second challenge the investigators found during their learning [**T1**]. If the queue size is not set properly and one provides a high publishing rate, messages can be lost. On the other hand, if the publishing rate

```
# Instantiate the node
rospy.init_node('robot_move')

# Create the publisher
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

# Create the message
message = Twist()
message.linear = Vector3(0.5, 0, 0)

# Define the rate
rate = rospy.Rate(10)

# Publish the speed at fixed rate of 10 Hz
while not rospy.is_shutdown():
    pub.publish(message)
    rate.sleep()
```

Listing 1: Code example of a publisher implemented in ROS.

```
sensor = list()

# Callback for scan topic
def scan_cb(scan_msg):
  sensor = scan_msg.ranges

# Callback for odometry topic
def odom_cb(msg):
  position:Pose = msg.pose.pose.position
  if sensor[90] < 1.0:
    print(f"Robot close to wall: {sensor[90]}")
    print(f"Robot Position: {position}")

# Subscribers definition
odom_sub = rospy.Subscriber('/odom', Odometry, odom_cb)
scan_sub = rospy.Subscriber('/scan', LaserScan, scan_cb)
rospy.spin()
```

Listing 2: An example concurrency issue.

is too small, the robot will struggle to move correctly. The queue size, the publishing rate, and the robot velocity being published are highly dependent on each other and impact the simulation, which to new users may be unexpected. For instance, a user may want the robot to stop immediately after a certain number of messages have been sent. However, the robot stop may not be immediate as it can still have messages in its queue, or the robot itself may have momentum.

Investigators also found another issue related to the control of the subscription rate. If one publishes information at 20Hz, and a subscriber intends to receive information at 10Hz, what should be the ideal method to throttle the subscriber that improves the robot performance.

3.5 Domain Knowledge on Robotic Systems

3.5.1 Message Content Impact in ROS (●●●).

While the previous challenge was related to the frequency of messages in the different ROS nodes, there is the additional issue of how to estimate and understand the impact of the message content with the real-world behavior. For instance, in the previous example, how the velocity value published affects the real speed of the robot is non-linear. A similar scenario occurred when trying to smoothly land a drone. However, taking into account the frequency of messages and their content is not enough to move the robotic system closer to the ground at a progressively lower speed. To have a better understanding, it is necessary to understand differential drive and specific implementation details, such as publishing frequency and likelihood of dropping messages. The abstraction model of ROS hides most of these details, hindering the connection between high-level code and its impact in the simulation or world.

3.6 Common Programming Errors

Every framework has common programming errors typically associated with its technical details. In this section, we identify two errors experienced by the investigators related to stringly-typed topic names (Section 3.6.1) and with the messages and services file names (Section 3.6.2).

3.6.1 Topic Identifiers (●●○).

ROS allows the programmer to subscribe and publish information to topics by providing the topic name as a string and the type of message it receives. The most common mistake during development was the mistyping of topic names (e.g., **\odom** instead of **/odom**) [**T2**, **T5**, **T13**]. Since no verification of identifiers is done in ROS, the system compiles and runs, but does not behave as intended.

3.6.2 Service Message Files (●○○).

Messages and services can have the same name. However, if both are used in the same node, the system emits errors that are not easy to trace back to the different entities having the same name.

Another problem is that having entities with the same name causes confusion for the developer, and the wrong one may be used in the wrong place [**T11**].

3.7 Concurrency

3.7.1 Shared Memory in Callbacks (●○○).

In the case where a node subscribes to two different topics and wants to publish to a third topic, the investigators found concurrency related issues that were addressed properly by neither the ROS API nor The Construct Sim.

In particular, consider the case where two callbacks are needed to handle the subscribed messages. There is a concurrency problem when these two callbacks read and write from the same shared memory.

Listing 2 defines two subscribers: Upon receiving a message, the callback for the **scan_sub** subscriber sets a shared variable with information from sensors. The sensors publish information to the **scan** topic, which provides information on the distance the robot is from an object. The callback for the second subscriber, **odom_sub**, receives the current position of the robot and checks whether there is an obstacle ahead; if so, the robot's position is printed.

The primary challenge is dealing with the concurrency on the sensor variable accessed in the **odom_cb** and **odom_cb**. This problem appeared when working in task **T22**. Consider that each callback is triggered at the rate at which messages are published to their respective topics. If **scan_sub** is called more frequently than **odom_sub**, the sensor information accessed at line 15 may already

be outdated. This race condition can lead to an unintended of the robotic system. A solution implemented by the investigators was the addition of a mutex to the sensor variable.

Despite the usefulness in ensuring the correctness of the information, the introduction of the mutex may change the frequency that the callback operates. However, changing the frequency at which the callback processes messages can have unpredicted side effects when running the robotic system.

3.7.2 Message Loss (●●○).

A common problem faced by the investigators was the loss of messages, which led to the robotic systems to idle [**T13**]. When a node that uses actions or services is launched and the corresponding action or service server is not ready yet, the published messages will be silently lost. This problem is only evidenced by the wrong behavior of the robot.

A similar problem occurs when a publisher publishes a topic only once but before the subscriber is listening. Nevertheless, although not initially apparent to the investigators, ROS allows the persistence of the last published message to a topic by "latching" the connection. If the user does not latch the connection, the order in which the subscriber and publisher are initiated matters. In both cases, the system requires the programmer to implement a waiting system to ensure that both ends are available for communication.

4 FUTURE DIRECTIONS

This report presented the experience of three investigators when introduced to the Robot Operating System. Firstly, the investigators followed the ROS Basics (Python) Course from The Construct Sim's training course. Then, they annotated the challenges and problems identified at each task during their learning. Finally, each challenge and the associated task was categorized and described at the end of the course.

We presented nine challenges and problems identified by the investigators and depicted in a mind map. The challenges were related to the build system, the ROS interface and its architecture, common programming errors, and concurrency issues. Based on the challenges that we identified, we describe several opportunities to improve the ROS developer experience for newcomers below.

Firstly, we address *Standard Methods* (Challenge 3.3.1). We hope that the presented results help design more in-depth usability studies with larger study groups. For instance, one could evaluate the difficulty of applying good practices in ROS and its impact on the robot's behavior. This challenge also motivates the introduction and improvement of verification tools, such as HAROS [8] and ROSDiscover [9], to ensure good coding practices and program correctness.

Secondly, we address the points presented in *ROS IDL Discoverability* (Challenge 3.2.1), *Message Loss* (Challenge 3.7.2) and *Message Content Impact in ROS* (Challenge 3.5.1), where we encourage the need for better documentation for each component. At a minimum, this should describe the component's interface, its intended communication model, and the frequency and bounds on messages values. In addition, better documentation for off-the-shelf components makes it easier to understand the roles of each component and how the user may adapt them to achieve the desired behavior.

Thirdly, we address the points in *Topic Identifiers* (Challenge 3.6.1) and *Service Message Files* (Challenge 3.6.2). This study motivates the introduction of novel frameworks and techniques that statically check the correctness of ROS systems and address the stated challenges. For instance, verifying ROS-specific properties, such as the stringly-typed topic names in the publications and subscriptions, could help prevent this category of errors. In particular, Dependent Types, where the expected type of messages is dependent on the topic name, can be useful to prevent such errors.

Finally, we address the points in *IDL File Consistency* (Challenge 3.1.1) and *Publisher-Subscriber Frequency Impact* (Challenge 3.4.1). We hope to motivate the analysis of the architecture of the robot and systems configuration files to provide novice and expert users with the information needed to correct an existing problem in their system. For instance, ROSDiscover already allows a certain level of analysis from the recovered systems architecture. We propose the introduction of techniques that allow the specification of the systems architecture by the user and the formal static verification of this architecture as a way to improve the system's correctness.

5 ACKNOWLEDGEMENTS

This work was supported by *Fundação para a Ciência e Tecnologia* (FCT) in the LASIGE Research Unit under the ref. (UIDB/00408/2020 and UIDP/00408/2020), and the CMU–Portugal Dual Degree PhD Program (SFRH/BD/151469/2021), by the CMU–Portugal project CAMELOT, (POCI-01-0247-FEDER-045915), the RAP project under the reference (EXPL/CCI-COM/1306/2021), and the U.S. Air Force Research Laboratory (#OSR-4066).

The authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

REFERENCES

[1] Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher Steven Timperley. 2021. Simulation for Robotics Test Automation: Developer Perspectives. In *International Conference on Software Testing (ICST '21)*. 263–274.
[2] 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 (ICST '20)*. 96–107.
[3] Gopika Ajaykumar, Maureen Steele, and Chien-Ming Huang. 2022. A Survey on End-User Robot Programming. *ACM Comput. Surv.* 54, 8 (2022), 164:1–164:36. https://doi.org/10.1145/3466819
[4] Robotic Systems Lab Legged Robotics at ETH Zürich. 2021. ROS Best Practices. https://github.com/leggedrobotics/ros_best_practices/wiki.
[5] Nathan P. Koenig and Andrew Howard. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2149–2154. https://doi.org/10.1109/IROS.2004.1389727
[6] 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* 3, 3.2 (01 2009), 1–6.
[7] Fanny Riedo. 2015. Thymio a holistic approach to designing accessible educational robots. (2015). https://doi.org/10.5075/epfl-thesis-6557
[8] André Santos, Alcino Cunha, and Nuno Macedo. 2021. The High-Assurance ROS Framework. In *3rd IEEE/ACM International Workshop on Robotics Software Engineering, RoSE@ICSE 2021*. IEEE, 37–40. https://doi.org/10.1109/RoSE52553.2021.00013
[9] Christopher S. Timperley, Tobias Dürschmid, Bradley Schmerl, David Garlan, and Claire Le Goues. 2022. ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems. In *International Conference on Software Architecture (ICSA '22)*. (To appear.).
[10] Keenan Wyrobek. 2017. The Origin Story of ROS, the Linux of Robotics. *IEEE Spectrum* (Oct 2017). https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics