Architectural Bug Detection and Analysis in ROS-based Mobile Manipulator Projects Using ROSDiscover

AZMYIN M. KAMAL* and LINTA ISLAM*, Louisiana State University, USA

With the recent advancements made in AI foundational models, cheap-yet-flexible sensory suits and ultra-fast IoT devices, mobile robotics have grown in complexity, functionalities and sophistication. Many such mobile platforms now incorporates multi-DOF robotic arms that gives it dexterity and precision for performing intricate tasks with remarkable efficiency. At the heart of this technology is the reusable software stack called the Robot Operating System (ROS).

While ROS provides composition of system architecture through its late-binding schema, well-defined core API and quasi-static architecture, it is prone to architectural configurations that may lead to fatal errors during runtime. Such errors dubbed, Architectural Misconfiguration Bugs are difficult to detect specially in large-scale ROS projects, typical for mobile manipulators.

In this report, we use study the run-time architectural errors detection using a Python-based static analyzer called ROSDiscover. We test this analysis tool to identify the architectural bugs in two small-scale ROS packages that uses common message and topic configuration found in mobile manipulator systems. From the experimental study, we report that, ROSDiscover is adept in detecting dangling subscriber/publisher pairs when remapping occurs at launch file but fails to detect fatal message mismatch configuration errors when such misconfiguration occurs in source files without any remapping.

Additional Key Words and Phrases: Run-time architecture, Static Analysis, ROS-ecosystem

ACM Reference Format:
Azmyin M. Kamal and Linta Islam. 2024. Architectural Bug Detection and Analysis in ROS-based Mobile Manipulator Projects Using ROSDiscover. *J. ACM* 37, 4, Article 111 (August 2024), 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Robotics is the field that encompasses the study and application of designing, building, operating, and using robots. The primary objective of robotics is to develop devices capable of aiding and supporting humans. Several robots are specifically designed to do tasks that pose a risk to human safety [12]; others are used to substitute individuals in tedious, monotonous, or unpleasant occupations [2]. Recently, robotics has seen significant growth due to ongoing technological advancements.

Robot Operating System (ROS or ros) is a freely available middleware suite for robotics. ROS is not an operating system, but rather a collection of software frameworks for developing robot software [27]. It offers services specifically designed for a diverse computer cluster, including hardware abstraction, control of low-level devices, implementation of commonly used functions, exchanging processes through message-passing, and management of software packages [1].

*Both authors contributed equally to this research.

Authors' Contact Information: Azmyin M. Kamal, akamal4@lsu.edu; Linta Islam, lislam4@lsu.edu, Louisiana State University, Baton Rouge, Louisiana, USA.

ACM 1557-735X/2024/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

The main objective of ROS is to facilitate the reuse of code in the field of robotics research and development. Many robotics research organizations and enterprises have rapidly embraced it as their standard development methodology [25]. Several completely developed robots, including humanoid robots like PR-2 [3] and REEM-C [16], ground robots like the ClearPath platforms [9], and aerial platforms from Ascending Technologies [10], are implemented using the Robot Operating System (ROS).

Architectural bugs refer to defects or errors in the design and structure of ROS-based systems [31]. These bugs often appear as issues with the communication and interaction patterns among different components of a ROS system, including nodes, topics, services, and parameters. Typical scenarios include mismatches in message formats, incorrect setup of node connections, or ineffective data management approaches that result in delays, data loss, or inability to perform desired actions. Due to ROS's modular architecture, where components work simultaneously and rely on each other, identifying and fixing these architectural bugs can be especially difficult [21]. This requires the use of advanced tools and methods to ensure the system's integrity and functionality.

ROSDiscover is an advanced software analysis tool created to improve the reliability and endurance of robotic applications that automate the complex task of architectural reconstruction for ROS systems [30]. Using static analysis methods, it carefully examines, analyzes, and understands the original source code of ROS nodes, thus revealing a system's architecture's structural composition and interconnections. This tool is essential for viewing and detecting any architectural bugs that may otherwise go undiscovered using conventional testing techniques. ROSDiscover allows developers to tackle inefficiencies and errors in design proactively. The release of ROSDiscover represents notable progress in the realm of robotics, offering developers a potent tool to guarantee the efficiency and accuracy of their systems.

Identifying architectural bugs in ROS is crucial for maintaining robots' effectiveness, dependability, security, and versatility [24]. Architectural bugs may lead to system malfunctions, inefficient use of resources, and hazardous actions, especially in human engagement settings. Early detection of these software defects is crucial for preserving system stability and maximizing performance, hence improving safety and resource allocation [18]. Finally, ensuring the structural integrity of ROS-based systems enhances the confidence of both developers and users, developing trust in the dependability and effectiveness of autonomous robotic systems.

Motivated by the above, our project aims to extend ROSDiscover to detect architectural bugs for ROS-1 systems involving mobile manipulators working in indoor environments. The objectives for this project are as follows:

(1) Develop two new small-scale ROS 1 projects using common message types for bug dataset creation.
(2) Perform qualitative and quantitative study on ROSDiscover's bug detection capability with the new dataset thereby ascertaining its feasibility of use with mobile manipulator ROS systems.

This report is organized into following sections. Section presents 2 discusses briefly about architectural misconfiguration bug discussion, about ROS and ROSDiscover. Section 3 illustrates the proposed methodology. Section 5 provides the result and analysis of our proposed methodology. and Section 6 concludes the report.

2 BACKGROUND

In this section, we first review the fundamental concept of ROS and its challenges. After that, we discussed about the architectural bugs in ROS, and why it is necessary to detect those bugs. Finally, we explained how ROSDiscover can solve the issues related to architectural bugs.

2.1 ROS

The concept of ROS was developed to meet the need for a universally applicable software platform in the field of robotics research and development. Over time, ROS has transformed from a specialized framework mostly used in academic environments to a widely recognized standard. The development of ROS has produced notable improvements, notably with the launch of ROS 2 in 2017 [17]. This update addressed some limitations of the original ROS, such as the ability to handle real-time data and enhanced security measures.

The standardized development and interface of robotic applications has been a major impact on ROS in both academia [23] and industry [8]. The capacity to include a wide range of hardware and software has established it as the fundamental support system for several research endeavors and commercial uses in different fields, such as self-driving cars [11], industrial automation [6], and personal robotics [15]. The open-source nature of ROS, together with its strong community, facilitates innovation by enabling researchers to use each other's work and explore the limits of what can be achieved in robotics.

ROS also provides difficulties in terms of real-time features. To overcome these problems, ROS 2 can be adopted which provides enhanced real-time support [19]. Not only this, ROS2 supports security, which was not provided in ROS1. Previously, multirobot communication was not possible due to lack of scalability. This issue has also been resolved in ROS-2.

2.2 Architectural Bug Detection in ROS

The architecture of a ROS-based system comprises several interconnected nodes and components that manage various operations, including sensing, and data processing. ROS has several types of bugs, such as incorrect message types, synchronization issues, and improper node connections. These bugs can have big effects, like system failures, unpredictable behavior, or wasteful resource use [29]. The complexity increases in mobile manipulators, where the synchronization between movement and manipulating motions is crucial. Identifying these bugs is important not only for the system's functional accuracy but also for its efficiency and dependability.

There are several challenges in identifying architectural flaws in ROS-based systems. First, maintaining a consistent system design is made more difficult by the dynamic nature of these systems, where components may be added or withdrawn [14]. In addition, the lack of synchronization in communication between nodes might result in unpredictable behaviors, causing problems to occur randomly and making them difficult to replicate [4]. Dealing with dependencies between different packages and nodes may also result in conflicts or unexpected behaviors if improperly handled. For the above-mentioned reasons, it is necessary to address the challenges of ROS-based systems.

2.3 ROSDiscover

ROSDiscover solves the above-described difficulties by offering a platform for the automated architectural study of systems based on ROS [30]. It uses static analysis methods to recreate the architecture of ROS applications based on their source code. This reconstruction enables the identification of not just the components and their interactions, but also possible architectural mismatches and inefficiencies. ROSDiscover examines the ROS computation network, detecting and analyzing the relationships between nodes, topics, services, and parameters. This feature may be also used to visually represent the structure of the system and identify specific regions that are vulnerable to errors. The tool can automate a procedure that was previously done manually and was prone to errors is a notable progress in the industry.

The use of ROSDiscover in real-world scenarios provides encouraging outcomes. ROSDiscover improves the dependability and security of mobile manipulators by automating the bug identification and system analysis processes. It promotes more effective methodologies for system design and deployment, surpassing the scope of conventional bug detection. Studies show that ROSDiscover takes less time and resources to find and fix design bugs. Thus, both the speed of development and the reliability of the system are improved after using ROSDiscover.

3 METHODOLOGY

In this section, we breifly discuss the theoretical framework and methodology of ROSDiscover. Figure 1 illustrates the overview of the analysis process. The analysis comprises three sections: Component Model Recovery, System Architecture Composition, and Bug Detection. They are briefly discussed below.

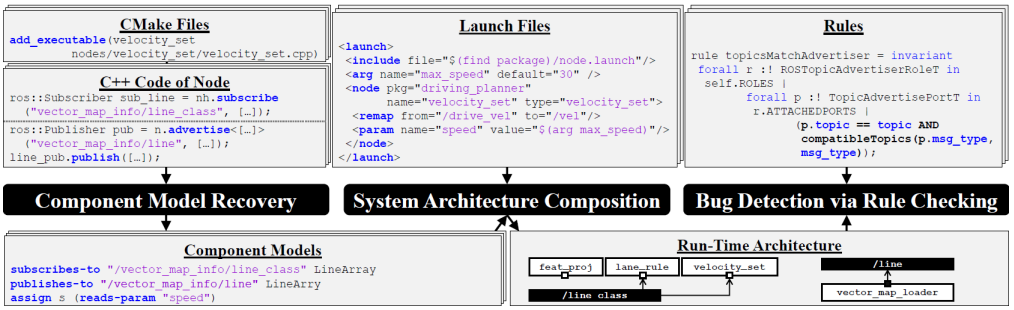

Fig. 1. Overview of ROSDiscover [30]

3.1 Component Model Recovery

In this stage, ROSDiscover recovers comprehensive architectural models from source code, launch and Cmake files. This stage involves reviewing the source code of ROS nodes in order to detect and describe the ROS API calls [26] that establish the interface of each component. ROSDiscover specifically searches for API calls associated with publishers, subscribers, service servers, and clients, since they are the determining factors for how nodes connect inside the ROS ecosystem. ROSDiscover captures these interactions to construct **component** models for each relevant class (i.e. 'line_class' from Autorally) that is represented as sets of symbolic function summaries.

3.2 System Architecture Composition and Bug Detection

In this phase, system architecture files such as **launch.xml files, Cmake** files, are recursively parsed to build the **component models** extracted previous step, into **component-connector-port** model for the whole system. All the models are then concatenated to form the **system architecture** represented in the ROS Acme style, a derivative from the Acme [7] format. An illustrative example is shown in Figure 2. Using this system architecture, rule violations may be detected in form of **architectural bugs**. Two critical bugs, relevant to this study, are listed below

(1) Connectors and ports cannot have **mismatched message types**. Such misconfiguration may lead to fatal system crash. For example a component **FetchSensor** configured to receive **std_msgs::String** cannot accept message of type **sensor_msgs::Imu** even if build passes compilation.

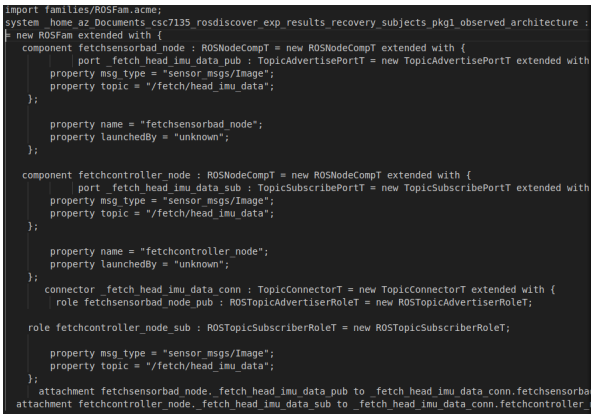

Fig. 2. Illustrative example of a the **component-connector-port** model in ROS Acme format for **pkg1**.

(2) **Dangling publishers/subscribers** i.e. components connecting to topics that produces or consumes no data are allowed. This architectural error is one of the major cause of undefined behavior in complex ROS system and very difficult to trace since, often they do not cause any crash during runtime.

4 DATASET,EXPERIMENTAL SETUP AND COLLABORATION

As discussed in Section 3, ROSDiscover aims to find architectural bugs in ROS systems which may constitute many packages. However, in the ROSDiscover paper, the authors reported that no bugs were possible to generate for the Fetch Manipulation system. Hence, to perform a feasiblity study of whether ROSDiscover, in its current state is suitable for use with mobile manipulator projects, we developed two packages to test its capabilties. Due to time constraint and the difficultly in finding a large ROS system involving a mobile manipulator and compatible with the ROS version we are using, we opted to develop two smaller ROS systems utilizing **std_msgs::String, sensor_msgs::Image, sensor_msgs::Imu and sensor_msgs::Imu**, the most common message primitives used in Fetch mobile manipulator projects. We call them **pkg1** and **pkg2** respectively. Both are written in C++ and tested to pass build for both correct and buggy architectures. Figure 3 and Figure 4 depicts the schematics of the architectures of the two packages where boxes in red represents a Subscriber and boxes in blue represents a Publisher. Flow of data is shown with **bold black** arrows with the type of message type shown on top. Erroneous type of data set (Figure 3) or dangling publishers/subscribers (Figure 4) are shown in red.

A brief description of the two packages, followed by the experimental setup and collaboration approach taken by the authors are discussed below

4.1 Dataset

Figure 3 depicts the correct and erroneous architecture for **pkg1**. Here *FetchController* is a class emulating the controls the motion of the Fetch robot's head (refer Figure 7) whilst *FetchSensorBad* emulates a class that computes the 6 DOF pose of the head using an onboard IMU. The bug generated in this package is designed to test if ROSDiscover is successful in detecting incorrect use of message type, a bug that in this case, leads to fatal behavior where *FetchController* is dropped from the ROS execution graph. Figure 5 illustrates this behavior when run on both Docker image and in the local host.

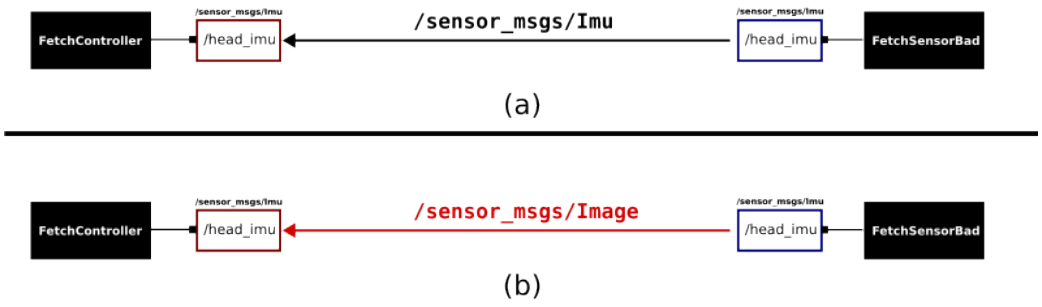

Fig. 3. Pkg 1 architectures. Architecture (a) is the correct composition while (b) is an erroneous composition that is caused by message type misinterpretation not caused during startup. Zoom in for better view. Best viewed in color.

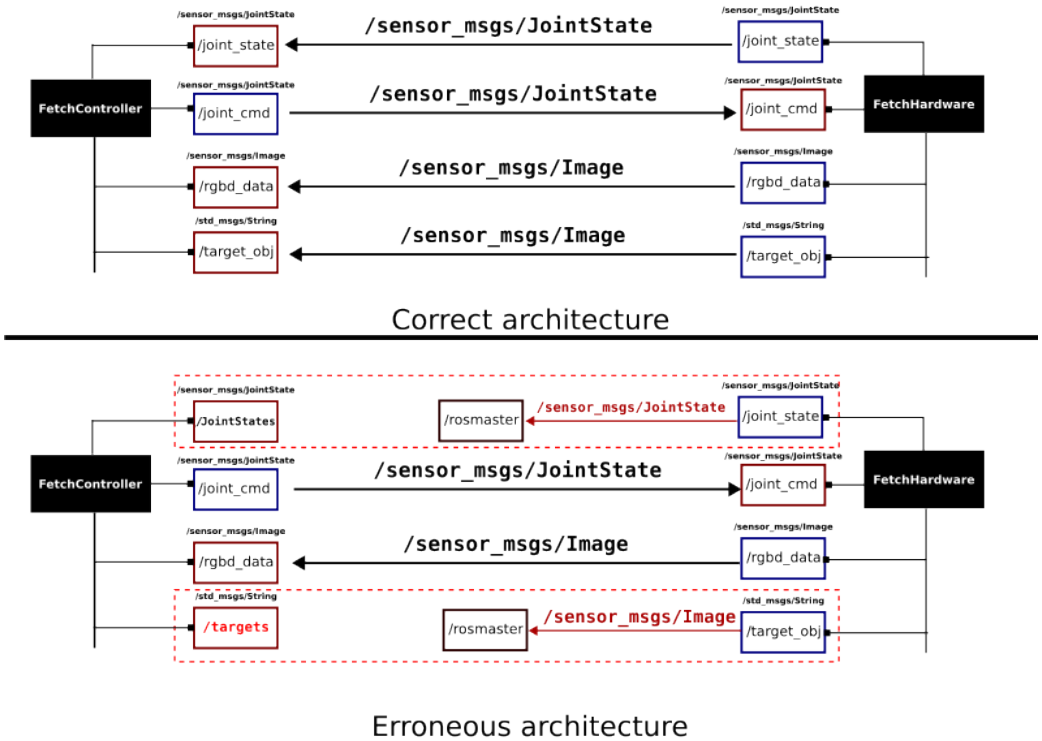

Fig. 4. Pkg 2 architectures. Architectural bugs in form of dangling connectors are shown with a red dashed rectangle. Zoom in for better view. Best viewed in color.

Figures 4 depicts a more complicated scenario where we emulate a Fetch manipulator grasping colored cube based on user specification. A Gazebo [13] illustration of this setup is shown in Figure 7.

```
FetchSensorBad: Sent IMU data
-------------------------
-------------------------
FetchSensorBad: Sent IMU data
-------------------------
[ERROR] [1714935126.870282994]: Client [/fetchcontroller_node] wants topic /
/head_imu_data to have datatype/md5sum [sensor_msgs/Imu/6a62c6daae103f4ff57a
f95cec2], but our version has [sensor_msgs/Image/060021388200f6f0f447d0fcd9c
]. Dropping connection.
-------------------------
FetchSensorBad: Sent IMU data
-------------------------
```

Fig. 5. System response for flawed **pkg1** architecture.

```
----------------------------------------
FetchController: target object: Red
FetchController: received joint state feedback
----------------------------------------
FetchHardware: received joint command data
----------------------------------------
FetchController: target object: Red
FetchController: received joint state feedback
----------------------------------------
FetchHardware: received joint command data
----------------------------------------
```

```
----------------------------------------
FetchHardware: received joint command data
----------------------------------------
FetchController: target object: Blue
----------------------------------------
FetchHardware: received joint command data
----------------------------------------
FetchController: target object: Blue
----------------------------------------
FetchHardware: received joint command data
----------------------------------------
```

Fig. 6. System response for **pkg2**. Left shows response for correct architecture where user wants the robot to manipulate the red cube, and right shows response for the erroneous architecture where user's command is bypassed and robot manipulates only the blue cube with no feedback on where its end effector is in the task space.

In this system *FetchController* takes in the RGBD data and class name of the object detected and emulates sending a series of joint command to the *FetchHardware* class. Two dangling publisher/subscriber pairs are generated that results in an unwarranted behavior where

(1) user's choice of cube is bypassed
(2) no feedback to joint states is received that may result in the manipulator colliding with the table surface.

Figure 6 illustrate the good and erroneous behavior for **pkg2**.

4.2 Experimental Setup and Collaboration

A ROS-Noetic [22] workspace was prepared and the two C++ packages described above were built and tested using catkin_tools [28], a toolchain designed to handle ROS packages. Then a Docker [20] image of the whole workspace is created. Through ROSWire [5], a tool that can manipulate ROS-specfic files in a Docker container, we use ROSDiscover to perform these tasks, a **modified procedure** based on the steps in the replication package[1] from the ROSDiscover paper.

(1) Setup an **experiments.yml** file that instructs ROSDiscover to use the desired launch file to start the two packages.
(2) **Observe** the system play out for 50 seconds and then repeat this 10 times
(3) **Generate** ROS-Acme style architecture
(4) **Compare** this against the 19 misconfiguration bug rules written in ROSDiscover
(5) Tabulate the **number of bugs** and **severity of bugs** reported and compare with ground-truths.

[1]https://zenodo.org/records/5834633

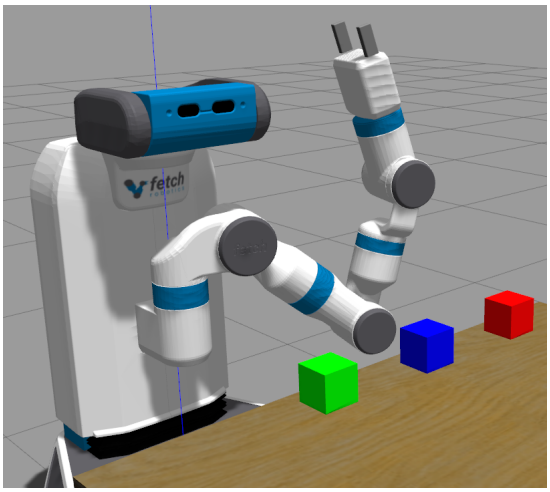

Fig. 7. Gazebo emulation of **pkg2** scenario.

All experiments were performed on a Ubuntu 20.04 machine equipped with a Ryzen 5600x processor, 16 GB ram and RTX 3080 GPU. The GPU was not used in this experiment. Git was used to track local changes and GitHub remote repositories were used. We utilized two repositories: **csc7135_proj_sp24**[2], for the catkin workspace, and **csc7135_rosdiscover_exp**[3], for ROSDiscover experiments. The authors envisioned collaboration as developing a software in a small team and mostly used pull requests, merge from development branches and reviews as shown illustrated in Figure 8. No code, without review, was merged into the **main** branch.

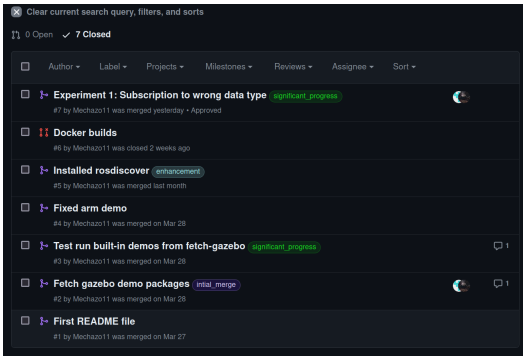

Fig. 8. Pull requests showing how collaboration was used for **csc7135_proj_sp24** repository.

5 EVALUATION AND RESULTS

To perform the proposed analysis, we formulate this research question

- **RQ:** *Can ROSDiscover detect severe architectural bugs and classify them in context of mobile manipulator projects?*

[2]https://github.com/Mechazo11/csc7135_proj_sp24/blob/main/README.md
[3]https://github.com/Mechazo11/csc7135_rosdiscover_exp

Table 1. Performance on ROSDiscover on the test dataset

Package name	Number of Bugs	Severity	Number of Bugs Detected	Severity reported	Accuracy (%)
pkg1	1	Fatal, runtime crash	0	No issue	**0%**
pkg2	4	Severe, unwarranted robot behavior	4	No comment on severity	**100%**

The philosophy behind this question is the fact that **pkg1** and **pkg2** represents two simple systems constructed only using *Topics* and common ROS **message** types. ROSDiscover on the other hand, was extensively test for much larger robotics project involving the core ROS architectural elements viz. *Topics, Services and Actions*. Thus we hypothesize 100% accuracy on the two test ROS packages.

However, Table 1 depicts an interesting result. For **pkg1**, ROSDiscover not only does not detect the fatal bug, but also reports no architecture errors. We attribute this to the limitation of the component model recovered since during runtime neither *FetchController* nor *FetchSensorBad* has their topics remapped. However, bug detection step appears to be fully functional, evident when we manually correct the acme file, the error prompt generated matches expected outcome. The actual and expected outcomes are shown in Figure 9. This finding corroborates the comment by the authors of ROSDiscover who stated that the *System Architecture* composition step gives highest priority to the **launch.xml**, followed by **Cmake** and then the **source files**. This appears to be a design tradeoff as evident from the paper where the authors state that most topic misconfiguration bugs in ROS systems were caused by **topic remapping**. However, in the case of **pkg1**, there is no remapping done during runtime, the launch file simply starts two nodes.

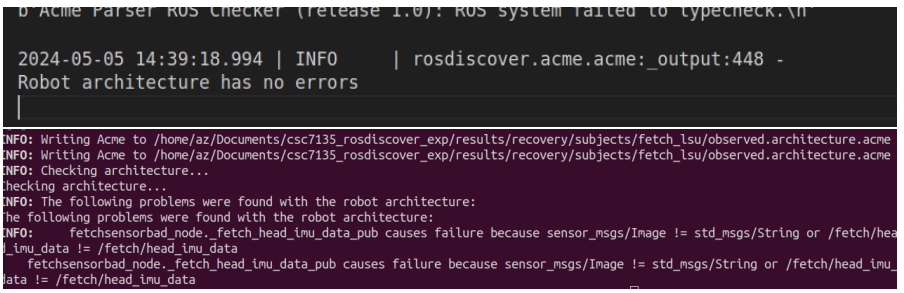

Fig. 9. ROSDiscover actual output (top) vs expected (bottom) for **pkg1**

.

Figure 10 shows the error analysis for **pkg2**. We observe that all four dangling publisher/subscribers have been detected. However, the system does not classify the severity of the error.

From the above, we conclude that ROSDiscover can indeed detect the architectural errors involving dangling subscribers/publishers provided topics with similar message type exists and remapping occurs in the launch files. However, it when errors rising from misuse of message type directly inside the node source files, it is unable to detect the architectural misconfiguration.

```
2024-05-05 14:36:02.466 | INFO     | rosdiscover.acme.acme:_output:448 -
fetchcontroller_node subscribes to an unpublished topic: '/fetch/targets'.
But there is a publisher(s) fetchhardware_node that publishes
a similar message type: std_msgs/String as /fetch/target_obj. fetchcontroller_node was
launched by unknown

2024-05-05 14:36:02.466 | INFO     | rosdiscover.acme.acme:_output:448 -
fetchcontroller_node subscribes to an unpublished topic: '/fetch/JointStates'.
But there is a publisher(s) fetchhardware_node that
publishes a similar message type: sensor_msgs/JointState as /fetch/joint_state.
fetchcontroller_node was launched by unknown

2024-05-05 14:36:02.466 | INFO     | rosdiscover.acme.acme:_output:448 -
fetchhardware_node  publishes to an unsubscribed topic: '/fetch/joint_state'.
But there is a subscriber(s) fetchcontroller_node._fetch_JointStates_sub with
a similar name that subscribes to a similar message type. fetchhardware_node was launched
from unknown.

2024-05-05 14:36:02.466 | INFO     | rosdiscover.acme.acme:_output:448 -
fetchhardware_node  publishes to an unsubscribed topic: '/fetch/target_obj'.
But there is a subscriber(s) fetchcontroller_node._fetch_targets_sub
with a similar name that subscribes to a similar message type.
fetchhardware_node was launched from unknown.
```

Fig. 10. ROSDiscover architecture analysis for **pkg2**

6 CONCLUSION

In this study, we have used ROSDiscover to find the architectural bugs in two small-scale ROS Noetic packages inspired by the common message and system architecture used in the Fetch mobile manipulator. The architectural recovery appears to depend on instances of topic remapping, which is significantly effective in identifying all the dangling publisher/subscriber pairs but did not demonstrate adequate performance when message type misconfiguration were done in-situ in the source files. In addition, ROSDiscover capability to classify severity of error appears to be limited and highly dependent on the 19 misconfiguration error database introduced in the original paper.

There were several challenges we faced while working with ROSDiscover. First, the steps provided for RQ3 and RQ2 in the replication package of ROSDiscover cannot be followed faithfully. This issue arose because we needed to use the **rosdiscover-cxx-extract** package, which lacked sufficient documentation for troubleshooting at the time of writing this report. Moreover, the **pipenv** provided in the ROSDiscover paper couldn't successfully install ROSDiscover and ROSWire, necessitating separate installations via another **pipenv** environment configured from the ROSDiscover copy available in its repository[4]. Additionally, testing some of ROSDiscover's results required Docker images totaling well over **100GB**, and the image repository recommended in the replication package had a broken download link when this report was written. These issues collectively caused significant delays in our project.

In the future, ROSDiscover can be more flexible if it can be decoupled from its current dependency on ROSWire and Docker. Additionally, adding a severity classification module can significantly improve the efficacy of this tool. Finally, modifying ROSDiscover with the ROS2 Core API will enhance its functions, which can be open many new avenus of research.

REFERENCES

[1] Adnan Ademovic. 2016. An introduction to robot OS: Toptal®. https://www.toptal.com/robotics/introduction-to-robot-operating-system

[2] Teodor Akinfiev, Manuel Armada, and Samir Nabulsi. 2009. Climbing cleaning robot for vertical surfaces. *Industrial Robot: An International Journal* 36, 4 (2009), 352–357.

[4]https://github.com/cmu-rss-lab/rosdiscover

[3] Jonathan Bohren, Radu Bogdan Rusu, E Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. 2011. Towards autonomous robotic butlers: Lessons learned with the PR2. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 5568–5575.

[4] 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 *Proceedings of the 4th International Workshop on Robotics Software Engineering*. 33–38.

[5] Carnegie Mellon University Robotics Institute. Year the repository was last updated or accessed. ROSWire: A Tool for Efficient Data Exchange between ROS 1 and ROS 2. https://github.com/cmu-rss-lab/roswire.

[6] Rinat Galin and Roman Meshcheryakov. 2019. Automation and robotics in the context of Industry 4.0: the shift to collaborative robots. In *IOP Conference Series: Materials Science and Engineering*, Vol. 537. IOP Publishing, 032073.

[7] David Garlan, Robert Monroe, and David Wile. 2000. Acme: Architectural description of component-based systems. (2000).

[8] Ruchi Goel and Pooja Gupta. 2020. Robotics and industry 4.0. *A Roadmap to Industry 4.0: Smart Production, Sharp Business and Sustainable Development* (2020), 157–169.

[9] Stephen J Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. 2009. Clearpath: highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 177–187.

[10] Norman Hallermann, Guido Morgenthal, and Volker Rodehorst. 2015. Unmanned aerial systems (uas)–case studies of vision based monitoring of ageing structures. In *International Symposium Non-Destructive Testing in Civil Engineering (NDT-CE)*. 15–17.

[11] Sertac Karaman, Ariel Anders, Michael Boulet, Jane Connor, Kenneth Gregson, Winter Guerra, Owen Guldner, Mubarik Mohamoud, Brian Plancher, Robert Shin, et al. 2017. Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at MIT. In *2017 IEEE integrated STEM education conference (ISEC)*. IEEE, 195–203.

[12] Albert WY Ko and Henry YK Lau. 2009. Intelligent robot-assisted humanitarian search and rescue system. *International Journal of Advanced Robotic Systems* 6, 2 (2009), 12.

[13] Nathan Koenig and Andrew Howard. 2004. Design and implementation of the player/stage/gazebo project: a generic approach to robot simulation. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*. IEEE, 3312–3319.

[14] Anis Koubâa et al. 2017. *Robot Operating System (ROS)*. Vol. 1. Springer.

[15] Anis Koubâa, Mohamed-Foued Sriti, Yasir Javed, Maram Alajlan, Basit Qureshi, Fatma Ellouze, and Abdelrahman Mahmoud. 2016. Turtlebot at office: A service-oriented software architecture for personal assistant robots using ros. In *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE, 270–276.

[16] Leonardo Lanari, Seth Hutchinson, and Luca Marchionni. 2014. Boundedness issues in planning of locomotion trajectories for biped robots. In *2014 IEEE-RAS International Conference on Humanoid Robots*. IEEE, 951–958.

[17] 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* 7, 66 (2022), eabm6074.

[18] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. 2020. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2718–2725.

[19] Steve Macenski, Tom Moore, David V Lu, Alexey Merzlyakov, and Michael Ferguson. 2023. From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2. *Robotics and Autonomous Systems* 168 (2023), 104493.

[20] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[21] Eduardo Munera, Jose-Luis Poza-Lujan, Juan-Luis Posadas-Yague, Jose Simo, and J Francisco Blanes Noguera. 2017. Distributed real-time control architecture for ROS-based modular robots. *IFAC-PapersOnLine* 50, 1 (2017), 11233–11238.

[22] Open Robotics. 2022. *ROS (Robot Operating System)*. https://www.ros.org/

[23] Elena Ospennikova, Michael Ershov, and Ivan Iljin. 2015. Educational robotics as an inovative educational technology. *Procedia-Social and Behavioral Sciences* 214 (2015), 18–26.

[24] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[25] Morgan Quigley, Brian Gerkey, and William D Smart. 2015. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.".

[26] André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves Dos Santos. 2017. Mining the usage patterns of ROS primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3855–3860.

[27] Daniel Serrano. 2011. Introduction to ROS-Robot Operating System.

[28] Catkin Tools Development Team. 2022. Catkin Tools: Command-line Tools for Building Catkin Workspaces. (2022). https://catkin-tools.readthedocs.io/
[29] Christopher Timperley and A Wąsowski. 2019. 188 ROS bugs later: Where do we go from here. *ROSCON'19* (2019).
[30] 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 *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, 112–123.
[31] Thomas Witte and Matthias Tichy. 2018. Checking consistency of robot software architectures in ROS. In *Proceedings of the 1st International Workshop on Robotics Software Engineering*. 1–8.