2024 IEEE International Conference on Robotics and Automation (ICRA 2024)
May 13-17, 2024. Yokohama, Japan

Is it a Bug? Understanding Physical Unit Mismatches in Robot Software

Paulo Canelas[†Φ], Trenton Tabor[†], John-Paul Ore[‡],
Alcides Fonseca[Φ], Claire Le Goues[†], and Christopher S. Timperley[†]

***Abstract*— Robot software is abundant with variables that represent real-world physical units (e.g., meters, seconds). Operations over different units (e.g., adding meters and seconds) may be incorrect and can lead to dangerous system misbehaviors; manually detecting such mistakes is challenging. Current software analysis techniques identify such mismatches using dimensional analysis rules and ROS-specific assumptions to analyze the source code. However, these are ignorant of the fact that physical unit mismatches in robotics code are often intentional (e.g., when operating a differential drive robot), resulting in false positive bug reports that can impede robotics developer trust and productivity. In this work, we study how developers introduce physical unit mismatches by manually inspecting 180 errors detected by the software analysis technique, Phys. We identify three types of physical unit mismatches and present a taxonomy of eight high-level categories of how these errors manifest. We find that developers often make unforced and paradigmatic physical unit mismatches through differential drives, small angle approximations, and controls. We draw insights on current development to inform future research to better detect, categorize, and address meaningful physical unit mismatches.**

I. Introduction

Robot code abounds with variables that are quantified using physical units representing real-world measurements. Ensuring the consistency of computations with these variables is critical to prevent systems from misbehaving. Physical unit mismatches arise when developers perform incorrect operations according to dimensional analysis [2]. For instance, adding meters with seconds is an incorrect operation.

To free developers from the burden of locating physical unit mismatches, software analysis tools, such as PhrikyUnits [3], Phys [4], and SA4U [5], help detect unit mismatches through dimensional analysis. For instance, Phys checks for incorrect unit assignments or mismatches in the arithmetic operation of units. Equation (1) presents the formula for physical unit arithmetic where given the multiplication of two variables, each in the form $(u_1^a, u_2^b, ..., u_n^m)$, with units (u_n) and their powers (u^m), the result is the sum of the powers of the base units.

$$(u_1^a, \ldots, u_n^x) \times (u_1^b, \ldots, u_n^y) = (u_1^{a+b}, \ldots, u_n^{x+y}) \quad (1)$$

*This work was supported by Fundação para a Ciência e Tecnologia (FCT) in the LASIGE Research Unit under the ref. (UIDB/00408/2020, UIDP/00408/2020 and EXPL/CCI-COM/1306/2021 [1]), the scholarship (SFRH/BD/151469/2021), and NSF-USDA-NIFA #2021-67021-33451. The authors would like to thank Bogdan Vasilescu for his feedback on this work.

[†]School of Computer Science, Carnegie Mellon University. {pasantos,ttabor,clegoues,ctimperl}@andrew.cmu.edu

[‡]North Carolina State University. jwore@ncsu.edu

[Φ]LASIGE, Faculdade de Ciências da Universidade de Lisboa. {pacsantos,amfonseca}@ciencias.ulisboa.pt

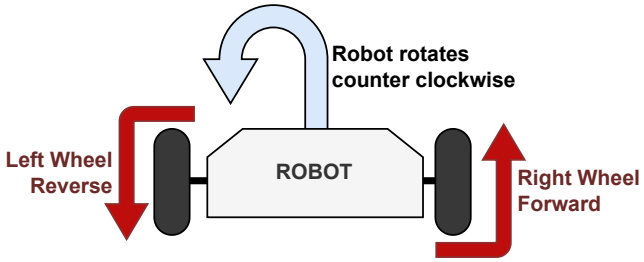

Fig. 1: Example of a differential robot where the linear and angular velocities control the system's direction and speed.

```
left_vel = msg.linear.x - msg.angular.z;
right_vel = -msg.linear.x - msg.angular.z;
```

Code Example 1: Physical unit mismatch of a differential drive mobile robot that subtracts linear (m/s) and angular velocities (rad/s).

All units and their powers must be the same for all variables when performing additions, subtractions, comparisons, and assignments. Given two variables, with units u_1 and u_2, the following unit operations are considered valid.

$$\begin{aligned} &u_1 + u_2 \ \textbf{when} \ \{u_1 = u_2\} \\ &u_1 > u_2 \ \textbf{when} \ \{u_1 = u_2\} \\ &u_1 := u_2 \ \textbf{when} \ \{u_1 = u_2\} \end{aligned} \quad (2)$$

Techniques also consider extra information to help improve their analysis. For example, SA4U detects unit errors in unmanned aerial vehicles by using the execution information of the robot to detect dimensional inconsistencies (e.g., millimeters vs. meters). PhrikyUnits and Phys are built on top of the Robot Operating System (ROS) [6] and exploit conventions regarding ROS messages and their units to determine the physical units involved in the operations. Furthermore, they use natural language processing on variable names to infer their unit. For instance, a variable named vel_x likely represents a velocity (with high confidence) in m/s.

When robot source code contains physical unit mismatches, these techniques apply their conventions and analysis rules to detect them. For instance, Code Example 1 contains a unit mismatch that calculates left and right wheel velocities for a differential drive robot (Figure 1). For this system, the speed of both wheels impacts the linear (m/s) and angular (rad/s) velocities. When encountering two operations, such as those in lines 1 and 2, whose unit types are rad/s and m/s, with the second operation having opposite

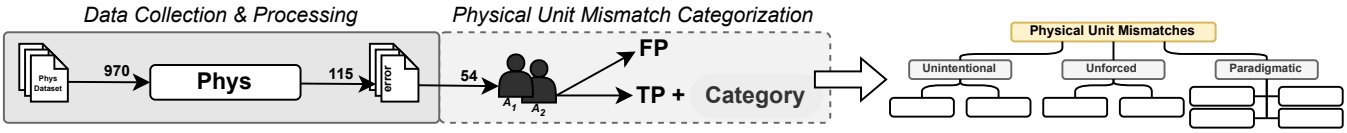

Fig. 2: Methodology followed to study the categories of physical unit mismatches in two steps. **Step 1.** Executes Phys in its dataset and collects a report of the the unit mismatches detected. **Step 2.** Two authors (A_1 and A_2) manually inspect each file, project, and error to determine if it corresponds to a unit mismatch and the mismatch type. Finally, both authors meet and collaboratively devise the final taxonomy.

signs, we confidently identify a differential drive. Introducing a physical unit mismatch in this system is inevitable to perform the differential drive. We refer to intentional and unavoidable domain unit mismatches as **paradigmatical**. Despite being an intentional mismatch by the developer, current analysis techniques raise an error when analyzing this system. Tools such as PhrikyUnits, Phys, and SA4U report these paradigmatical mismatches as bugs that require the developer's attention. The increased number of irrelevant positive detections hinders the trust and adoption of these analysis techniques by developers [7], [8]. Therefore, to make these tools effective in practice, we must understand where mismatches occur and when they are paradigmatical.

In this work, we set out to understand the categories of physical unit mismatches in ROS software and how these intentional unit errors manifest. Previous research quantitatively analyzed the frequency of units, messages, and types of arithmetic operations involved in physical unit mismatches [9]. However, to the best of our knowledge, qualitative studies have yet to analyze the different kinds of mismatches and how they manifest. We fill this knowledge gap with the following contributions:

1) A taxonomy of physical unit mismatches with detailed description and examples from real-world scenarios;
2) A discussion of our findings and implications for future analysis techniques to detect unit mismatches.

II. Methodology

To understand the nature of physical unit mismatches in robot software, we address the following research question:

RQ: What types of physical unit mismatches do developers make in ROS-based robot software?

To answer this question, we used the dataset from Phys [4], a state-of-the-art technique for detecting physical unit mismatches. The Phys dataset is based on 28,484 `C++` unique files found on GitHub that use standard ROS Messages libraries (i.e., `geometry_msgs`). These come from various systems (e.g., ground, aerial, and underwater) for many purposes, like control, planning, communication, and navigation. Unlike the other analysis techniques, such as SA4U, Phys is a source-code level analysis tool that detects unit mismatches without requiring system execution or program traces, simplifying our analysis process. Our study methodology, illustrated in Figure 2, consists of two steps:

1. Data Collection & Processing. We gathered 970 C++ source code files from the Phys repository,[1] and instrumented Phys to obtain the low-confidence analysis information from the tool. Phys' execution on its dataset yielded 115 files with a total of 180 unit mismatches. Our dataset contains the analyzed code along with error messages from Phys.

2. Physical Unit Mismatch Categorization. Initially, two authors randomly selected 54 files each, half of the dataset, and manually inspected the error message and the respective source code. When unsure, the authors also analyzed the surrounding context of the contents in the file. Then, each author reviewed the descriptions from the other to prevent bias in the classification. During this validation step, the authors updated the classification on the categories of six of the files containing errors. This review resulted in updates to the description or categorization as true positive or false positive for each error. A true positive occurs when we effectively detect a unit mismatch, and false positive otherwise. Finally, we collaboratively devised a taxonomy to group each error considered an example of an actual unit mismatch.

Threats to Validity. While our study provides insights into the categories of physical unit mismatches, there are limitations to consider. Firstly, our analysis may be biased towards the inspected files from the Phys dataset. Although our analysis is based on 108 source code files, these represent real systems mined by Phys' authors from GitHub. Secondly, the authors' manual source code analysis may introduce human error. To prevent misinterpretations, each author reviewed the other's files. Thirdly, this study relies on Phys for detecting physical unit mismatches. Phys has several limitations, including that Phys is 1) *incomplete* because it infers physical units based on ROS Message libraries and variable names, and not all variable names are useful for inferring physical units; and, 2) *unsound* because it is not *flow-sensitive* and does not reason about *aliasing* (like `C++` pointers). These limitations mean that the Phys dataset contains physical unit mismatches that are detectable within these limitations. Any inherent limitations or biases in Phys' analysis can affect the completeness of the results. Nevertheless, we do not claim our taxonomy to be complete. This study presents the starting point for identifying unit mismatches, and we promote future studies to inspect and present any unidentified categories. Finally, this study is based on ROS-based code where some categories (e.g., message abuses) may not generalize to non-ROS software.

[1] `https://github.com/unl-nimbus-lab/phys/`

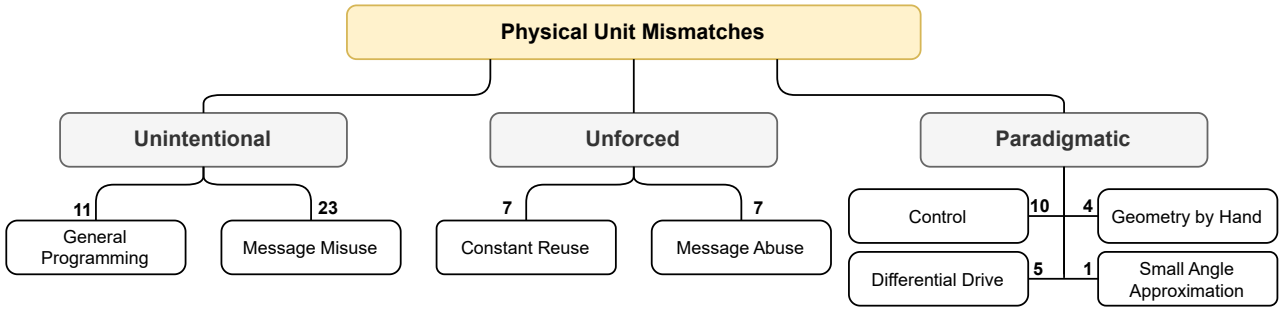

Fig. 3: Mindmap of the three high-level categories of physical unit mismatches, their sub-categories respective frequency.

```
vth = (dt == 0)? 0 : dth / dt;
// ...
odom_msg.twist.twist.angular.z = dth;
```

Code Example 2: A typo error from the Phys dataset where dth (rad) is assigned to angular.z (rad/s) rather than vth (rad/s).

```
diff_pose2D.x = end_pose2D.x-start_pose2D.x;
diff_pose2D.y = end_pose2D.y-start_pose2D.y;
difference.linear.x = diff_pose2D.x;
difference.linear.y = diff_pose2D.y;
```

Code Example 3: Message misuse where the difference of two poses (m) is assigned to a linear velocity (m/s).

III. Taxonomy of Physical Unit Mismatches

This section presents the categories of physical unit mismatches we defined during our study. We provide a high-level categorization of each mismatch as "Unintentional", "Unforced", or "Paradigmatic" depending on the intuition on whether the developer intended to perform the mismatch, and if it is avoidable in the robotics domain. For example, copy/paste errors or using the wrong data structure in a way that does not provide some benefit are Unintentional Mismatches. Mismatches indicating knowledge about the mismatch (e.g., a comment) that are avoidable, such as a bad practice, are classified as Unforced mismatches. Finally, mismatches that are inherent in an algorithm or standard operation necessary for the execution of the system are Paradigmatic Mismatches.

We present examples of real-world physical unit mismatches representative of the bugs encountered on the Phys dataset. Figure 3 presents the mindmap with the eight categories of unit mismatches and their respective frequency.

A. Unintentional Mismatches

Intuitively known as *bugs*, these are examples of source code parts where developers have an incorrect knowledge of the code's execution or a misunderstanding of ROS conventions. These are the most common motivation of analysis tools.

General Programming. This category encompasses mismatches that occur as a result of general programming mistakes. Tools such as Phys, PhrikyUnits, and SA4U are explicitly designed to identify these mismatches. Our analysis identifies four subcategories of general programming mistakes that lead to unintended mismatches.

a) Duplication: These mismatches occur when the developer duplicates the same statement into a different program context where variables with the same names use different units. These usually happen in if-statements where multiple ways exist to interact with the system. By copying and pasting code from different places without adapting it, the developer ends up using variables and values that may have different units in the destination, leading to a physical unit mismatch. For example, this occurred when trying to compare linear and angular velocities after duplicating previous expressions in the Phys dataset.[2]

b) Typo: Developers typically name variables according to their meaning (e.g., pos_x, pos_y, pos_z). Since a given function may contain multiple variables whose names differ by one or two characters, a small typo easily allows developers to unwarily use the wrong variable. Code Example 2 presents a unit mismatch due to a typo. In practice, such a mistake can cause the robot to turn aggressively, which is especially dangerous for larger robots.

c) Assignments: Mistakes also occur when a variable with the wrong physical units is incorrectly assigned to a variable. This error occurs when assigning the wrong physical quantity (e.g., Code Example 2) or when using the wrong units for the same physical quantity (e.g., radians vs. degrees) due to a missing or incorrect conversion. For example, a large dth in Code Example 2 impacts the system rotation, unexpectedly having a heavy robot sharply turning.

d) Additive Operations: These errors occur when variables with different physical units are added or subtracted, violating the dimensional analysis rules. Such errors can occur due to using the wrong operator (e.g., due to a typo) or a misunderstanding of the units involved.

e) Comparison Operations: Like *Additive Operators*, these errors occur when the rules of dimensional analysis are violated as a result of comparing variables with different

[2]robot_trajectory_node.cpp#L116

```
max_velocity = 0.1;

if (cmd.linear.x > max_velocity)
    cmd.linear.x = max_velocity;
// ...
if (cmd.angular.z > max_velocity)
    cmd.angular.z = max_velocity;
```

Code Example 4: max_velocity is reused to represent both linear (m/s) and angular velocities (rad/s).

physical units (e.g., $=$, $\leq$, $>$). For example, during our analysis we identified a mistake within the visualization code where linear and angular velocities were incorrectly compared when deciding which results to plot.[3]

Message Misuse. Mismatches also occur within ROS when an inappropriate message format, intended to represent a given physical quantity, is misused to share data of a different physical quantity. In such cases, the developer could have used a different standard ROS message format, intended for the physical quantity, or otherwise defined their own. Within our dataset, most misuses occurred when using Pose, Point, and Twist messages in lieu of one another. This behavior is common enough to appear on robotics software forums, such as the ROS Reddit.[4]

Code Example 3 shows a common message misuse mistake. The developer assigns the difference of two Pose messages, each representing the robot's position and orientation, to a Twist message, meant to represent velocity and subsequently treated it as a relative position. This quantity would have been more appropriately represented by a Pose message. Message misuses may not be problematic as long as they are known and accounted for. However, they may eventually lead to mistakes when components (e.g., written by different developers or at a later date) rely on the format of the messages to understand their meaning.

B. Unforced Mismatches

Unforced mismatches are made with avoidable developer intent, possibly with lost efficiency. These can include "code smells" or patterns considered bad practice in the robotics domain, but they are not intuitively considered *bugs*. In the Phys dataset, most instances relate to the developer saving a few bytes or trying to manage fewer data structures.

Constant Reuse. Unit mismatches can also occur when a single constant is intentionally used throughout a program to represent several, distinct physical quantities. Overloading the meaning of a given variable negatively impacts the readability of the code, makes it harder to diagnose problems, and creates an opportunity to easily introduce a defect (e.g., when the developer wants to update one meaning of the variable but not all meanings).[5]

[3] twist_marker.cpp#L65

[4] https://www.reddit.com/r/ROS/comments/j6deqo/

[5] placement_wrt_workspace_action_server.cpp#L130

```
odom_msg.pose.pose.position.z = th_;
```

Code Example 5: Message abuse where a Pose message position field (m) carries extra information, th_ (rad), about the robot on a 2D plane.

For example, in Code Example 4, the developers use a single floating point constant, max_velocity, to store an offset for both linear (v) and angular velocity (ω). While this mismatch may be acceptable to the developer if the offsets are coincidentally the same for both velocities, using the same variable to represent different errors or offsets is a bad practice. Consider the task of changing the maximum linear velocity of this system: In a large codebase, both linear and angular velocity comparisons and assignments may be separated by large chunks of code. Developers may mistakenly believe that changing the max_velocity assignment will work as intended, however, this will also impact the maximum angular velocity. For this reason, when defining error constants and offsets, a separate variable should be defined for each distinct physical quantity.

Message Abuse. We define Message Abuse as using a minority of the fields of a message incorrectly to pass extra data. Standard messages often have fields that are not required in a given application. A developer may choose to store related information in these fields, avoiding the need to send either a separate or a more complicated message. We identify a message abuse when the developer uses the incorrect unit in such fields consistently throughout a system. For instance, when abusing Pose messages in a 2D plane, developers take advantage of the z Quaternion field to transmit an angle rad (Code Example 5). In another example from the Phys dataset, the developer uses a Pose message to pack angles into the quaternion field, a misuse also seen on the official ROS Answers forum.[6]

Similar to Message Misuse, these mismatches can lead to problems with component reuse without careful documentation. However, since these fields are often ignored, problems may not surface until later development stages. If a developer produces a new component that interact with the current version, they may quickly introduce bugs by violating undocumented assumptions. Furthermore, if one intends to change the system later, it can become difficult to track which messages follow the implicit assumption and which do not.

C. Paradigmatic Mismatches

Robot software relies on standard algorithms and techniques from geometry, control, and other fields that often contain intentional but benign dimensional analysis violations. We refer to the mismatches that inherently result from these algorithms as *paradigmatic mismatches*, and consider them essential to developing robot software. Existing analysis techniques such as Phys and SA4U treat these mismatches in the same manner as unintentional and

[6] https://answers.ros.org/question/9400/

```
double robot_atan2 = atan2(
    robot_pose.getOrigin().y()+sin(robot_yaw),
    robot_pose.getOrigin().x()+cos(robot_yaw)
);
```

Code Example 6: A unit mismatch occuring due to manual geometry calculations in lieu of relying on a library.

```
angle_error = front_left - front_right;
speed.angular.z = BOUND(-0.1,
                        angle_error*angle_Kp,
                        0.1);
```

Code Example 7: A mismatch in a P controller pattern due to the possible assignment of an angle to an angular velocity.

unforced mismatches, potentially hampering developer trust in the analysis by alerting on known non-issues.

Geometry by Hand. These mismatches occur when geometry calculations are written from scratch instead of using a library. Code Example 6 illustrates such a mismatch. The code calculates the angle and direction of a robot's 2D position and orientation. The unit mismatch occurs due to the sum of the robot position with the yaw angle. Performing geometry by hand may contain operations involving multiple variables and sub-expressions, making them prone to errors.

Differential Drive. In a differential drive robot (e.g., Figure 1), one can rotate the body of the robot by combining linear and angular velocities. When specifically working with differential wheeled systems, combining both velocities is a standard practice, taught in any robotics class. Nevertheless, as this practice violates the dimensional rules, verification techniques incorrectly raise an error during their analysis.

As previously discussed, Code Example 1 presents the code for a mobile robot where the speeds of the left and right wheels (rad/s) influence both the robot's rotational (rad/s) and straight-line (m/s) motion. When we detect operations, such as those with unit types rad/s and m/s , especially when the second operation has opposite signs, it indicates a differential drive mechanism. This code pattern is relevant for improving analysis techniques, preventing them from falsely detecting unit mismatches, and helping developers focus on fixing unintentional mismatches. This practice occurs in introductory robotics courses, as well as across the official ROS answers page.[7,8]

Small Angle Approximation. There was one occurrence of a small angle [10] approximation in the dataset. When working with systems where compute budgets are tight, small angle approximations are used for speedup. A variable that contained an angle radian was used as a ratio unitless . Since for small angles, $\theta \approx sin(\theta)$ we considered this an example of an intentional mismatch.[9]

Control. As physical systems with dynamics, robot action requires control algorithms, often with a form like:

$$correction(\text{ units / second }) = f(error(\text{ units }))$$

Expressions of this form are always a unit mismatch, unless a developer chooses a function f that specifically scales the error by 1/sec . Most frequently in this dataset, controllers were P controllers, with or without saturation, like in Code Example 7. Similarly, we also found another example of exotic controllers in this data which manifested the same physical unit mismatch as the P controller.[10,11]

D. Analysis Limitations

We identify three limitations in analysis tools that can lead to incorrectly reporting or failing to report unit mismatches:

Physical Unit Inference. Automatically and correctly determining the physical unit behind a variable is essential to accurately idneitfying mismatches. Phys, for example, makes assumptions about commonly used ROS messages and variable naming conventions to predict variable physical unit types. However, Phys does not consider custom-created messages, which account for 15% of messages in the packages they analyze [11]. Furthermore, developers do not always use informative or accurate variable names (e.g., using width to define a scaling factor when it usually used for distance). To detect physical unit mismatches effectively, bug-finding techniques must also be robust to poorly-named variables.

Single File Analysis. Analysis techniques, such as Phys, only analyze single files. However, ROS-based systems rely on the interaction between multiple nodes in multiple files. When files are considered individually, analysis tools cannot distinguish between incorrect message assignments (Section III-A), messages abuses (Section III-B), and message misuses meaning there is intent behind these mismatches. However, the unit inconsistencies encountered in messages abuses and misuses are consistent throughout the system, and the interaction between different software components. If one intends to distinguish message abuses from unintended message assignments, it is critical to understand how message contents are used throughout the system.

Code Execution Path Sensitivity. We detected unit mismatches related to the reuse of variables depending on the mode the system is running. Code Example 8 presents an example of this physical unit mismatch. In this case, the developer switches between rotating and going forward depending on the value of x_forward. The physical unit mismatch occurs due to the assignment of multiple units to rotate and forward. However, during code execution, only a branch is executed, and the units are consistent on either path individually. These may be limitations of the analysis technique itself.

[7] https://answers.ros.org/question/197626/
[8] https://answers.ros.org/question/380434/
[9] virtual_trajectory_tracking.cpp
[10] eband_trajectory_controller.cpp#L526
[11] eband_local_planner.cpp#L1086

```
float x = m.linear.x;
float y = m.angular.z;

if (x_forward) {
    rotate = y;
    forward = x;
} else {
    rotate = x;
    forward = y;
}
```

Code Example 8: An example of a unit mismatch that is reported due to limitations in the underlying analysis.

IV. Discussion

Types of Physical Unit Mismatches. Our study identified three types of physical unit mismatches: unintentional, unenforced, and paradigmatic. Despite the previous assumptions regarding the analysis of physical unit mismatches, we found that not all of these are unintentional. Given the studied dataset, we detect that developers often unforcedly and pragmatically perform physical unit mismatches. Compared to the prior quantitative analysis [9], our qualitative work supports the author's findings related to the frequency of messages (e.g., Pose and Twist) and operations involved in unit mismatches. However, our qualitative analysis provides a deeper understanding of how these mismatches manifest. We determine the types of physical unit mismatches and the ways they manifest by providing a taxonomy of unit mismatches.

For each type of unit mismatch, we saw specific categories of mismatches that arise. Unintentional mismatches occur within two categories: when developers make general programming mistakes and misuse messages. Unforced mismatches happen when a developer uses a message field to transport extra information or reuses constant to compare linear and angular velocities. Paradigmatic physical unit mismatches arise when using robot-related concepts, such as differential drive, controls, geometry by hand, and small angle approximations. When developers create robot software, they purposely create these mismatches. However, introducing these mismatches in the robotics domain is correct, and techniques that raise errors hinder their adoption.

Improving Analysis Techniques. When detecting physical unit mismatches, it is essential to distinguish the different types and provide helpful error messages for developers to understand and fix their system quickly. Some specific categories of intentional physical unit mismatches present precise semantics in the source code, i.e., a pattern in the operations and units leads to a unit mismatch. For instance, differential drives are identifiable through consecutive operations involving `rad/s` and `m/s`, where the latter is the reversed signal. Constant reuses occur when the same variable is used in different unit operations, messages abuses, and misuses when there is a mismatch between ROS conventions and the user intent. In contrast, the overall units are consistent when interacting between the different parts of the system, and geometry by hand when we detect that developers manually perform geometry operations. Furthermore, different unit mismatches may have different levels of importance. While paradigmatic physical unit mismatches related to the robotics domain (e.g., differential drive and controls) should be ignored, unintentional mismatches should be immediately fixed as they may impact the system execution, and unforced mismatches can be presented as warnings for the developers that these represent errors that impact code maintainability. One can help developers focus on the next most important task by prioritizing the physical unit mismatches. Error messages are also critical for helping developers understand and fix the errors in their systems. Providing helpful error messages and warning them of the effects of their bad practice creates awareness of the impact of these mismatches.

Verification techniques must consider the domain when analyzing the source code to avoid raising irrelevant errors for developers. Unforced physical unit mismatches represent bad programming practices that hinder code maintainability. These mismatches introduce implicit undocumented assumptions on how the system works. Developers do not know the reasoning behind assigning different units to messages or reusing the same constant for different units. Furthermore, as the system grows and becomes more complex, introducing more code can break the previous assumptions and lead the system to an unknown behavior. Analysis techniques should raise a warning and distinguish the different physical unit mismatches. For instance, if one intends to distinguish between message abuses and unintended message assignments, checking how different ROS components interact is critical. To improve effectiveness in this detection, future tools should consider multiple files and components during analysis and verification. Techniques such as ROSDiscover [12] can help obtain the system's structure, such as the connections between nodes and topics, enabling the multi-file bug detection of physical unit mismatches. The presented types of unit mismatches define specific patterns that are helpful for future techniques to categorize these errors automatically.

V. Conclusion.

In this work, we uncovered the types of physical mismatches and verified how intentional unit mismatches manifest. We found that intentional unit errors occur due to the performing operations of the robotics domain, such as differential drive and controls, and also through bad programming practices by developers, through the abuse and misuse of messages, or reusing constants. We also found that a minority of the errors analyzed correspond to general programming errors, such as typos, duplication, and incorrect assignments. Given these insights, we promote the development that embeds the robots' domain in their analysis. Our study provides examples for each category of physical unit mismatch, along with a description of the mismatch. These can be used as a starting point for producing analysis to help developers detect and fix physical unit mismatches effectively.

REFERENCES

[1] A. Fonseca and P. Canelas, "Resource-Aware Programming," 2024. [Online]. Available: http://doi.org/10.54499/EXPL/CCI-COM/1306/2021

[2] P. Bridgman, *Dimensional Analysis*. Yale University Press, 1922.

[3] J. Ore, C. Detweiler, and S. G. Elbaum, "Phriky-units: a lightweight, annotation-free physical unit inconsistency detection tool," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 352–355.

[4] S. Kate, J. Ore, X. Zhang, S. G. Elbaum, and Z. Xu, "Phys: probabilistic physical unit assignment and inconsistency detection," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2018, pp. 563–573. [Online]. Available: https://doi.org/10.1145/3236024.3236035

[5] M. Taylor, J. Aurand, F. Qin, X. Wang, B. Henry, and X. Zhang, "SA4U: practical static analysis for unit type error detection," in *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2022, pp. 87:1–87:11.

[6] M. Quigley, K. Conle, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, pp. 1–6, 01 2009.

[7] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 332–343.

[8] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM (CACM)*, vol. 61 Issue 4, pp. 58–66, 2018. [Online]. Available: https://dl.acm.org/citation.cfm?id=3188720

[9] J. Ore, S. G. Elbaum, and C. Detweiler, "Dimensional inconsistencies in code and ROS messages: A study of 5.9m lines of code," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2017, pp. 712–718.

[10] O. Amidi and C. E. Thorpe, "Integrated mobile robot control," in *Mobile Robots V*, W. H. Chun and W. J. Wolfe, Eds., vol. 1388, International Society for Optics and Photonics. SPIE, 1991, pp. 504 – 523. [Online]. Available: https://doi.org/10.1117/12.25494

[11] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos, "Mining the usage patterns of ROS primitives," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2017, pp. 3855–3860.

[12] C. S. Timperley, T. Dürschmid, B. R. Schmerl, D. Garlan, and C. Le Goues, "Rosdiscover: Statically detecting run-time architecture misconfigurations in robotics systems," in *19th IEEE International Conference on Software Architecture, ICSA*. IEEE, 2022, pp. 112–123.