2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 24–28, 2017, Vancouver, BC, Canada

Dimensional Inconsistencies in Code and ROS Messages: a Study of 5.9M Lines of Code

John-Paul Ore, Sebastian Elbaum, and Carrick Detweiler

***Abstract*— This work presents a study of robot software using the Robot Operating System (ROS), focusing on detecting inconsistencies in physical unit manipulation. We discuss how dimensional analysis, the rules governing how physical quantities are combined, can be used to detect inconsistencies in robot software that are otherwise difficult to detect. Using a corpus of ROS software with 5.9M lines of code, we measure the frequency of these dimensional inconsistencies and find them in 6% (211 / 3,484) of repositories that use ROS. We find that the inconsistency type 'Assigning multiple units to a variable' accounts for 75% of inconsistencies in ROS code. We identify the ROS classes and physical units most likely to be involved with dimensional inconsistencies, and find that the ROS Message type *geometry_msgs::Twist* is involved in over half of all inconsistencies and is used by developers in ways contrary to *Twist*'s intent. We further analyze the frequency of physical units used in ROS programs as a proxy for assessing how developers use ROS, and discuss the practical implications of our results including how to detect and avoid these inconsistencies.**

I. Introduction

The increasing sophistication of the robots we build has raised the complexity of the software that drives these robots. Part of that complexity is caused by the required integration of quantities measured in physical units into the code. Consider the 3,484 repositories of the Robot Operating System (ROS) [1] code we study in this work. These repositories have hundreds of thousands of program points where variables represent physical quantities including time, distance, angles, torques, teslas, and others.

These physical units have meanings and rules on how they can be manipulated in the physical world. Yet those meanings and rules are not always apparent in the code, leading to hazards known as dimensional inconsistencies [2].

These inconsistencies can become faults that are not detected by the compiler or the build tools. For example, Fig. 1, line 30 incorrectly implements the computation of a Euclidean distance. Although the compiler will not detect this fault, dimensional analysis of the physical units reveals that the units of the squared term `err_x * err_x` is meters-squared (m^2) and the units of the addition `err_y + err_y` results in meters (m). Furthermore, this code might successfully pass tests on the robot because it can be correct (accidental correctness as per the test input selection) or almost correct (weaker version of a test oracle), making it difficult to detect the fault until the deployed robot does something notoriously wrong.

```
26 void goal_d(double x_t,double y_t, double t)
27 {
28     err_x=x_t-X;
29     err_y=y_t-Y;
30     err_d=sqrt(err_x*err_x+err_y+err_y);
```

Fig. 1: Addition of inconsistent units at line 30, `err_x*err_x` is meters-squared, `err_y+err_y` is meters.
source: `https://git.io/vytcm`

```
736 void callback (const geometry_msgs::Twist &msg) {
737 // TODO: fix this it is ugly!!
738 // (divide ground truth from GPS!!)
739     if (! enableAbsoluteError) {
740         current_position.x = msg->linear.x;
741         current_position.y = msg->linear.y;
742         current_position.z = msg->linear.z;
743         }
744     desired_position.x = msg->angular.x;
745     desired_position.y = msg->angular.y;
746     desired_position.z = msg->angular.z;
747     }
```

Fig. 2: Inconsistent usage of ROS Message *Twist*, designed for linear and angular velocities, instead used for positions in lines 740-746. Comment from source.
source: `https://git.io/vytCd`

Addressing dimensional inconsistencies has been studied in software engineering for decades [3], and many preventative measures have been proposed, such as native programming language support for units ($F\#$ [4]), specialized developer annotations [5], or specialized units libraries (*boost::units* [6]). Measuring the adoption rate for such efforts is difficult and beyond the scope of this work, but we observe that only 18 repositories contain *boost::units* and only two contain $F\#$ out of the 3,484 repositories. In part, we argue that the perception of, or the real burden associated with, the use of these approaches (e.g., annotate code, migrate code to use specialized libraries) hinders their adoption.

Instead, the robotic community seems to have taken a different approach, favoring and adopting frameworks like ROS[1] that standardize data structures and representations of commonly used physical quantities [7] in order to facilitate code reuse. For example, the ROS Message type *geometry_msgs::Wrench.torque.x* represents a physical quantity with units $kg\ m^2\ s^{-2}$.

Computer Science and Engineering, University of Nebraska-Lincoln, NE, 68588 USA {jore, elbaum, carrick}@cse.unl.edu
This work partially supported by NSF NRI-1638099, NSF CCF-1718040, USDA-NIFA 2013-67021-20947 and USDA-NIFA 2017-67021-25924.

[1]ROS has +3100 citations, monthly downloads of 8M packages and 1M web pageviews. Source: `http://wiki.ros.org/Metrics`

Unfortunately, the availability of messages with unit types does not guarantee their correct usage, even when the code compiles and builds. Such **physical unit inconsistencies hinder interoperability and maintainability.** The importance of standard message formats for interoperability in robot software is emphasized by Walck *et al.* [8] and Jung *et al.* [9]. For example, Fig. 2 shows a code snippet using a ROS Message of type *geometry_msgs::Twist*, which is defined to contain linear and angular velocities in 3-D space. The developer, however, uses this structure to carry two points in 3-D space (x, y, z positions) because the shape of two x, y, z points is conveniently the same shape as *Twist*. Note the developer's code comment in Fig. 2 acknowledging the need for additional software maintenance. This hinders portability, a goal of ROS.

In spite of the importance of such inconsistencies, as a community we do not yet have a good sense of how often and in what contexts such inconsistencies occur. In this work, we aim to provide initial answers to these questions, which requires at least two elements: an accessible robot code corpus that serves as a meaningful population for analysis, and an automated mechanism to detect dimensional inconsistencies that can scale to analyze such a large population. The recent upswing of popularity of robot programming and the widespread adoption of ROS, together with the availability of open source software has enabled us to construct a corpus of ROS code, the largest ROS corpus to date to our knowledge and the first study of these inconsistencies at scale. We have also recently developed a software analysis tool PhrikyUnits [10] for ROS C++ to find dimensional inconsistencies without code annotations or migration, which allows us to provide large-scale empirical evidence that the misuse of physical units exists in at least 6% of repositories we studied. PhrikyUnits is written in Python, open-source, and available for download at `https://github.com/unl-nimbus-lab/phriky-units`. In particular, we investigate the following research questions:

- **RQ1**: How frequently do dimensional inconsistencies occur in programs that use ROS?
- **RQ2**: What units are used in ROS, and what does this tell us about how ROS is used?
- **RQ3**: What ROS Message classes are most commonly used with incorrect units?

The contributions of this work are:

- A study of the units and dimensional inconsistencies found in a large-scale corpus of ROS software in 3,484 repositories, with 5.9M lines of code in files that contains ROS Messages.
- An analysis of the 357 inconsistencies found, including units and unit groups most frequently associated with inconsistencies.
- A case study of the *Twist* ROS Message type motivated by its frequent involvement in inconsistencies.
- A discussion of the practical implications of our findings, including how to detect and avoid these inconsistencies.

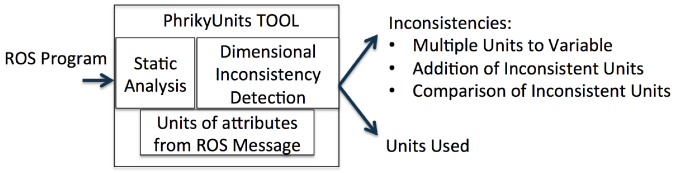

Fig. 3: Overview of how ROS programs are analyzed with the tool PhrikyUnits.

II. Background

In this section, we provide background on dimensional analysis and our software tool, PhrikyUnits[2], used to detect dimensional inconsistencies.

A. Dimensional Inconsistencies

The mathematical interaction of quantities representing physical phenomenon are governed by the rules of dimensional analysis [2]. Dimensions are measured in ROS using SI units [11]. Adding quantities with different dimensions results in a dimensional inconsistency, while multiplying combines the units of both quantities. We perform dimensional analysis but instantiate it within the framework and language of units, to use terms like 'force $kg\ m\ s^{-2}$' and 'torque $kg\ m^2\ s^{-2}$' that are easier for robot developers to understand rather than referring to force by it's dimensions: 'MASS * DISTANCE / (TIME * TIME)'.

Some quantities regularly used in robotics, such as radians, quaternions, and degrees, are unitless 'coherent units of measure' [12]. They interact as unitless scalars during multiplication but during addition they can *only* be added to quantities of the same type. This helps detect inconsistencies that add, for instance, radians with quaternions.

B. Detecting Inconsistencies with 'PhrikyUnits'

For this study, we used version 1.0 of our software analysis tool 'PhrikyUnits' for ROS C++, and the details of how PhrikyUnits works can be found in our previous work [10], [13]. PhrikyUnits is a completely automated static analysis tool that requires no developer annotations to detect inconsistencies. It is command-line driven and can be scripted and parallelized to analyze multiple files.

PhrikyUnits detects three different kinds of dimensional inconsistencies: 1) multiple units assigned to a variable (Fig. 2); 2) comparison of inconsistent units; and, 3) addition of inconsistent units (Fig. 1).

The high-level flow of PhrikyUnits is shown in Fig. 3. As shown in the figure, PhrikyUnits takes a ROS program as input, performs static analysis to assign physical units to variables based on their ROS Message type,[3] propagates unit information through the code across mathematical transformations, and detects dimensional inconsistencies. In the code example in Fig. 1, the variables `X` and `Y` had previously been assigned meters (not shown). The variable `X`

[2] `https://github.com/unl-nimbus-lab/phriky-units`

[3] PhrikyUnits uses a built-in map from message types to physical units based on the documentation of the ROS Message types, which explicitly specify units.

was assigned a value from *pose.position.x* of ROS Message type *geometry_msgs::Pose* and therefore has the units meters, likewise `Y`. During the subtraction on lines 28-29, meters is propagated to `err_x` and `err_y` by assignment. Then in line 30, `err_x` and `err_y` are added in a dimensionally inconsistent way, since `err_x` is squared while `err_y` is not. PhrikyUnits then reports the inconsistency type ('Addition of inconsistent units'), line number (30), and units involved (m^2 and m). PhrikyUnits can detect this dimensional inconsistency because ROS Messages provide unit information.

In terms of detection rate, we recently showed that PhrikyUnits has an 87% true positive rate [13]. Since not all variables containing physical units use ROS Messages (they can use types defined by the developer), and PhrikyUnits only recognizes those using ROS Messages, it will always underestimate the number of dimensional inconsistencies. Furthermore, by design, PhrikyUnits will only report inconsistencies when it can infer them with a high degree of certainty (all variables involved in the inconsistency must have an associated physical unit). For example, constants do not have an associated physical unit, so multiplying by a constant or unknown scalar reduces PhrikyUnits' degree of certainty, and PhrikyUnits will only report high-certainty inconsistencies by default. Therefore our results are an under approximation of the true number of dimensional inconsistencies in the corpus.

III. Related Work

There are several approaches to detecting or avoiding dimensional inconsistencies in software, including tools like Osprey [5] and UniFi [14], languages like $F\#$ [4], and libraries like *boost::units* [6]. Unfortunately, both Osprey and *boost::units* require annotations and Osprey only works on Java programs. In our corpus we find *boost::units* in only 0.5% (18 / 3,484) of repositories, and F# is only found in 2 repositories. UniFi works without annotations but uses the source code itself to infer inconsistencies and therefore requires both a correct and incorrect example within the same file, limiting generality. Therefore for dimensional inconsistency detection we rely on our own automatic tool PhrikyUnits. For more information on how software analysis is applied to robot software, see Cortesi *et al.* [15].

Santos *et al.* [16] analyzed a corpus of 50 ROS software repositories to assess code quality based on a variety of traditional software quality metrics. Likewise we seek to analyze software issues within the ROS community. Unlike their work, we analyze a much larger dataset of 3,484 repositories and focus on dimensional inconsistencies because robot software is threatened by this fault.

Ray *et al.* [17] studied a large-scale software corpus to assess the frequency of particular software faults across different programming languages. Like their work we use GitHub as the source of our corpus, mine software repositories, and make general conclusions about software usage. Unlike their work, we only look at C++, target ROS software, and focus on dimensional inconsistencies.

IV. Study Design

To address the research questions identified in Section I, we designed a study to apply our dimensional inconsistency and physical unit detection tool, PhrikyUnits, to a large-scale software corpus. In this section, we describe the methodology used to create the software corpus, and give technical details about how PhrikyUnits counts units and inconsistencies.

A. Software Corpus

We sought to build a corpus of ROS code with physical units specified by standard ROS message types, because ROS messages have attributes defined to have units, and because detecting dimensional inconsistencies requires units. GitHub is one of the largest collections of open-source code available and has been used as the basis of other large-scale software studies [17]. To find ROS code with units, we used the GitHub code search API to submit keyword queries for each ROS message type defined at `http://wiki.ros.org/common_msgs`, and extracted the repository names from the results. In total we found 4,736 repositories that contained search hits on ROS-related terms. Of this, 73% or 3,484 repositories contain compilable C++ code that uses the ROS messages defined to have physical units. Within these 3,484 repositories, we found a total of $20,843$ files with units containing $5,950,839$ lines of C++ code as measured using the tool CLOC (`http://cloc.sourceforge.net`). To our knowledge, this is the largest scale analysis of ROS source code to date. We provide a complete list of repositories used in this study at our GitHub repository.

The corpus contains duplicate code (approximately 30%) that we decided to leave in the corpus because we wanted to assess the frequency of units in code that is re-used across ROS developers.

B. Counting Units and ROS Class Usage

For this study, we modified PhrikyUnits to output the units of every variable it could identify, at every point these variables were read or written. We further modified PhrikyUnits to track the ROS Message classes involved with dimensional inconsistencies.

With this corpus and a tool to automatically detect dimensional inconsistencies, we ran PhrikyUnits on these 20,843 files and collated the results.

V. Results

In this section we begin by provide results and examining how frequently dimensional inconsistencies are found in the corpus. Next we examine and discuss the most frequently used units in ROS. Finally, we examine what ROS Message types are involved in these inconsistencies.

A. Frequency of Inconsistencies

Dimensional inconsistencies in software appear in several forms, and the most common in ROS is the 'Multiple units assigned to the same variable' type, as shown in Table I. This inconsistency represents 75% (267/357) of all inconsistencies found by our tool, and is mostly likely to occur with meters

INCONSISTENCY TYPE	COUNT	UNITS	MOST FREQUENT UNITS COUNTS
Multiple units assigned to the same variable	267	m	204
		$m\ s^{-1}$	171
		s^{-1}	71
		quaternion	30
		m^2	27
		radian	15
		$kg\ m\ s^{-2}$	4
Addition of inconsistent units	61	$m\ s^{-1}$	34
		m	32
		s^{-1}	14
		quaternion	10
		m^2	6
		radian	5
		$m^2\ s^{-2}$	1
Comparison of inconsistent units	29	$m\ s^{-1}$	21
		s^{-1}	6
		m	6
		m^2	4
		$m^2\ s^{-1}$	2
		s	1

TABLE I: Dimensional Inconsistencies by Type with the most frequently involved units. Note that multiple units can be involved with one inconsistency.

and meters-per-second, as shown in the table. The meters-squared associated with 'Addition of incompatible units' are usually caused by improperly formed distance metrics (Euclidean distances), like that shown in Fig. 1. These distance metrics are either typos or combinations of dissimilar units, which can behave correctly because of implicit constraints on the values that effectively normalize the values. However, these implicit assumptions hinder portability and might introduce faults when these assumptions change. The comparison of inconsistent units happens for a variety of reasons, but most often involve velocities and inconsistent interactions with time.

All inconsistency types were more likely to be caused by interactions between simple units, such as seconds, meters, meters-per-second, and quaternions. The more sophisticated units (combination of three or more base units) like torque are used less frequently in the corpus and account for an even smaller percentage of inconsistencies, suggesting that either the developers who work with sophisticated units are more careful not to cause dimensional inconsistencies, or the space for inconsistencies across those units is smaller. Further, many inconsistencies are caused when developers use ROS Message types contrary to their specification. This might not manifest as incorrect behavior if these misused data structures are used consistently, but causes confusion when sharing or maintaining code.

Overall, these inconsistencies were detected in 211 of the $3,484$ repositories, or 6%. This 6% answers RQ1, and this result shows that even with our underestimate, these kinds of problems lurk in a significant number of repositories.

B. Units Used and Frequencies

Table II shows the frequency of physical units used in ROS code. By 'Unit Usage by ROS Msg Definition' we mean the number of program points where a variable has units because it is a ROS Message attribute or the result of a known math operator, like `atan2`. By 'Unit inferred usage by assignment' we mean the number of program points where a variable has units not based on a ROS Message definition but instead inferred by the context of the program as the result of assignment statements and mathematical operations. This distinction is important because it tends to separate the units used externally in ROS Messages to communicate between nodes from those used internally in a ROS node during computation.

At a high level, Table II shows that simpler units are used more frequently, in more repos and files, and used more frequently during computations. There are some exceptions to this overall trend, including for meters-squared, force, torque, and radians, as we now discuss.

The radian unit, as shown in Table II, is the most common way to represent an angle, but notice that it is used more times as an inferred unit (21,557) than as a ROS Message definition (159). This suggests that robot software developers make extensive use of this representation of an angle, but that ROS does not have a standard way to represent it within ROS nodes. The radian's inferred usage comes mostly from the result of math operators such as `atan2, acos` or `asin`.

Force ($kg\ m\ s^{-2}$) is only found in 4% of repositories (154/3484), but is used 2,395 times. Likewise torque ($kg\ m^2\ s^{-2}$) is found in 7% of repositories (257/3484) and used 2,391 times. This means most ROS projects do not measure, compute, or communicate about forces and torques, or that many users are not using standard message types for force and torque. However, repositories that use force and torque perform several calculations and manipulations on these quantities. This might suggest that $< 10\%$ of ROS projects involve systems like robot arms, where force and torque measurements are more common.

Meters-squared (area or pose covariance) is used by definition 333 times and inferred 770 times. The inferred uses are usually Euclidean distance metrics, while the use by definition is position covariance. Although these quantities have the same units, they represent different kinds of quantities and should not be combined or compared, but in this case dimensional analysis would not detect this, because they have the same units.

These results address RQ2, and indicate that the more sophisticated units (like force and torque) are used in less than 10% of repositories, and that most ROS code achieves its goals using a combination of less complex units.

C. ROS Message Classes Most Likely to be Used with the Wrong Units.

PhrikyUnits detects when ROS Messages are used with units contrary to their specification, often the result of interactions between two conflicting sources of unit information. In our case, this interaction occurs because of a mismatch between the units specified by the ROS Message type, and the units actually assigned to the variables of the ROS Message.

UNIT NAME	SI UNIT	REPO COUNT	FILE COUNT	UNIT USAGE by ROS MSG DEFINITION	UNIT INFERRED USAGE by ASSIGNMENT
meter	m	2,669	9,930	112,538	19,525
second	s	2,433	9,939	85,299	9,573
quaternion (rotation)	*(dimensionless)*	2,078	6,169	49,449	2,749
angular velocity	s^{-1}	1,790	4,313	17,645	1,363
velocity	$m\ s^{-1}$	1,598	3,961	21,885	2,078
radian (angle)	*(dimensionless)*	1,106	3,133	159	21,557
acceleration	$m\ s^{-2}$	355	456	1,580	171
torque	$kg\ m^2\ s^{-2}$	257	403	2,373	18
area *or* pose covariance	m^2	187	314	333	770
degree 360 (angle)	*(dimensionless)*	172	232	844	68
angular acceleration	s^{-2}	168	199	544	3
acceleration covariance	$m^2\ s^{-4}$	156	183	495	0
Newton (force)	$kg\ m\ s^{-2}$	154	606	2,366	29
Tesla (magnetic induction)	$kg\ s^{-2}\ A^{-1}$	46	52	151	10
Celsius (temperature)	C°	37	40	42	2
Pascal (pressure)	$kg\ m^{-1}\ s^{-2}$	17	21	23	2
lux	lx	12	12	12	0
Pascal covariance	$kg^2\ m^{-2}\ s^{-4}$	3	3	3	0

TABLE II: Most common physical units used in 20,843 files across 3,484 open-source repositories in 5.9M lines of code, based on units from both ROS Messages and units inferred in the code by PhrikyUnits.

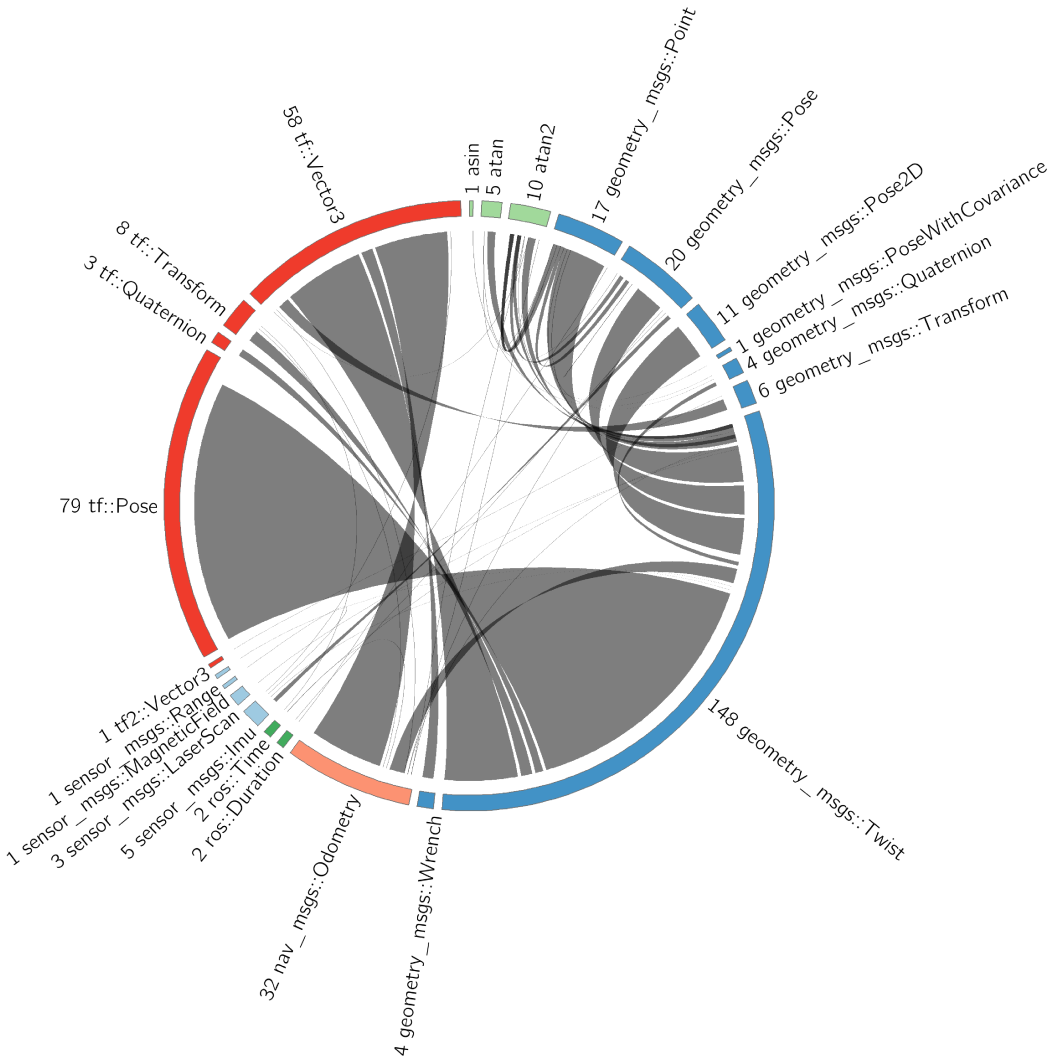

Fig. 4: Pairs of ROS Message classes involved with dimensional inconsistencies. Edges between ROS Message classes indicate an instance of inconsistent usage involving these two classes. Numbers preceding the ROS Message class indicate the number of inconsistencies. Stamped and unstamped messages were combined.

To help identify the ROS classes most likely to be used together inconsistently, we plotted the pairs of ROS Message classes involved in inconsistencies in Fig. 4. Note that this figure would not show dimensional inconsistencies such as those from Fig. 1 because that inconsistency only involved units that originated from one ROS Message class, *geometry_msgs::Pose*. This figure shows an edge drawn between classes to indicate a pairwise inconsistent interaction. For example, the inconsistent usage shown in Fig. 2 results in a edge between *geometry_msgs::Twist* and *geometry_msgs::Pose*. Some ROS Messages types have two subtypes, stamped and unstamped, which are identical other than a timestamp attribute. Fig. 4 combines stamped and unstamped messages for simplicity.

As shown in Fig. 4, usage of *geometry_msgs::Twist* accounts for 41% (148/357) of all inconsistent ROS Message usage, and is used most frequently in combination with *tf:Pose* and *tf::Vector3*. Also note the inconsistencies between *tf::Vector3* and *nav_msgs::Odometry*, that often happen with the velocity portion of Odometry, much in the same way as happens with *Twist*.

D. Limitations

Note that the reported frequencies of inconsistencies is a conservative underestimate, because not all variables have units associated to them by ROS Message types or by assignment, and further because of the limitations of this kind of software analysis technique, as discussed in Section IV.

PhrikyUnits runs in < 3 seconds on most files but we observed slower performance on very large auto-generated inverse kinematics files ($> 50MB$). Since these files contain almost no unit information from ROS Messages, to accelerate the analysis, we skipped them.

VI. Practical Implications

Use standardized ROS units. Our study found that standardized ROS units are used in 70% (3,484/4,736) of the accessed repositories, with units related to position, time, and velocity making the bulk of the units we identified (they are 2.4 times more common than the rest of the units combined). As mentioned, the usage estimate is an under-approximation, as many declared variables containing physical units do not employ the standardized ROS units. For example, we found that variables named 'time' and 'duration' are defined with type *ros::Time* or *ros::Duration* in 39% (4,123/10,530) of the instances those variable names are used, otherwise they do not have standardized ROS units that could be leveraged by our dimensional analysis. Not using standardized units negatively impacts reuse, making code comprehension more difficult, and undermining the application of tools like PhrikyUnits that can help to detect dimensional inconsistencies.

Run an automated checker to detect physical unit inconsistencies in code. Even a lightweight inconsistency detection tool like PhrikyUnits, which requires no additional effort for code annotation or migration, can detect certain physical unit inconsistencies with high confidence. On a MacBook Pro ('Early 2015') 2.9 GHz Intel i5 with 16GB of memory, it can analyze approximately 150 lines of code per second, its operation is trivially parallelizable, and it can be easily integrated as part of standard building processes. So, even for practitioners that have been hesitant to invest in code annotations or specialized libraries usage, there is little reason not to run a tool like PhrikyUnits.

Avoid common anti-patterns. Since *geometry_msgs::Twist* is the most misused ROS Message type, we performed an additional analysis of how *Twist* is used by ROS developers.

We modified PhrikyUnits to track assignments made to variables of type *Twist*. *Twist* has 6 attributes: 3 linear velocity components x, y, z and three angular velocity components x, y, z. For every *Twist* message in the corpus, we tracked which of these 6 attributes were written during programs, and the results are shown in Table III.

			twist.linear.			twist.angular.		
USAGE	TOTAL	COUNT	x	y	z	x	y	z
2-D planar	2,591	1,172	✓					✓
		1,101	✓	✓				✓
		201						✓
		117	✓					
3-D	1,534	1213	✓	✓	✓	✓	✓	✓
		169	✓	✓	✓			✓
		152	✓	✓	✓			

TABLE III: Usage of *geometry_msgs::Twist* showing majority of 2D planar usage of a 3D structure. A '✓' indicates an attribute was written, and a blank means the attribute was never written. Table does not show read-only instances.

As shown in Table III, *Twist* is mostly used for 2-D planar robots (2-D in this case means that the program never writes to attribute *linear.z*). This usage is not inconsistent in itself, since *Twist* is intentionally overloaded to mean either 2-D or 3-D velocities (Euclidean dimensions). However, many of these instances also use *angular.z* to store the heading, not angular velocity as intended. Further, we have observed *Twist* being often used not as a velocity but as a kind of 'delta', as shown in Figure 5. As shown in the figure, developers add the content of *Twist* directly to *Pose*. PhrikyUnits detects this dimensional inconsistency because the units do not match. Overall, Table III shows that *Twist* is used in many different and sometimes inconsistent ways, making it difficult for others consuming such messages to correctly interpret what *Twist* means. This might indicate the need to revisit the overload of the structure of this message.

VII. Conclusion

In this work we provided a characterization of the usage of physical units and the manipulations that are deemed dimensionally inconsistent in the code of robotic systems. We collected a corpus of 5.9M lines of ROS C++ code that uses ROS Messages representing physical quantities and we tailored a fully automated software analysis tool, PhrikyUnits, to assist in the unit recognition and in the detection of dimensional inconsistencies related to the unit usage.

```
222 // Calculate Deisred Position from Vel Cmd
223 desired_wp.position.x = current_gps_.pose.pose.position.x + cmd_vel.linear.x;
224 desired_wp.position.y = current_gps_.pose.pose.position.y + cmd_vel.linear.y;
225 desired_wp.position.z = desired_wp.position.z + cmd_vel.linear.z;
226 desired_wp.yaw = gps_yaw + cmd_vel.angular.z;
```

Fig. 5: *Twist* used incorrectly as a 'delta'. Variable `cmd_vel` is of type *Twist.*

source: `https://git.io/v7eMj`

We found that physical units represented by standard ROS message types are present in 73% of the repositories we studied, but most variables containing physical units do not employ the standard types. We also found that dimensional inconsistencies occur at least in 6% of the repositories we analyzed, including some highly recognized ones, and explained how this is a significant under approximation. Our investigation also revealed the most likely culprits, with *geometry_msgs::Twist* being at the top of the list in terms of being used inconsistently, contrary to its specification.

Our findings point to the need for more extensive use of standardized units to facilitate not just reuse but also dimensional inconsistency detection, for the use of cost-effective checkers like the one shown, and for further awareness about the risks of using certain standardized types that can be easily but erroneously overloaded.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3.2. Kobe, Japan, 2009, p. 5.

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

[3] M. Karr and D. B. Loveman III, "Incorporation of units into programming languages," *Communications of the ACM*, vol. 21, no. 5, pp. 385–391, 1978.

[4] A. Kennedy, "Types for units-of-measure: Theory and Practice," in *Central European Functional Programming School*. Springer, 2010, pp. 268–305.

[5] L. Jiang and Z. Su, "Osprey: a practical type system for validating dimensional unit correctness of c programs," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 262–271.

[6] M. C. Schabel and S. Watanabe, "Boost," *Units*, vol. 1, no. 0, pp. 2003–2010, 2008.

[7] Open Source Robotics Foundation, *ROS Enhancement Proposal 103*, July 2010 (accessed 25 Feb 2017). [Online]. Available: http://www.ros.org/reps/rep-0103.html

[8] G. Walck, U. Cupcic, T. O. Duran, and V. Perdereau, "A case study of ROS software re-usability for dexterous in-hand manipulation," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, 2014.

[9] M. Y. Jung, M. Balicki, A. Deguet, R. H. Taylor, and P. Kazanzides, "Lessons learned from the development of component-based medical robot systems," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 25–41, 2014.

[10] J. P. Ore, C. Detweiler, and S. 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*. ACM, 2017, pp. 352–355.

[11] I. Bureau of Weights, Measures, B. N. Taylor, and A. Thompson, "The International System Of Units (SI)," 2001.

[12] P. J. Mohr and W. D. Phillips, "Dimensionless units in the SI," *Metrologia*, vol. 52, no. 1, p. 40, 2014.

[13] J. P. Ore, C. Detweiler, and S. Elbaum, "Lightweight detection of physical unit inconsistencies without program annotations," in *Proceedings of the 2017 International Symposium on Software Testing and Analysis*. ACM, 2017, accepted, to Appear.

[14] S. Hangal and M. S. Lam, "Automatic dimension inference and checking for object-oriented programs," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 155–165.

[15] A. Cortesi, P. Ferrara, and N. Chaki, "Static analysis techniques for robotics software verification," in *Robotics (ISR), 2013 44th International Symposium on*. IEEE, 2013, pp. 1–6.

[16] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2016, pp. 4491–4496.

[17] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.