Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool

John-Paul Ore, Carrick Detweiler, Sebastian Elbaum
Computer Science and Computer Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska, USA 68588-0150
jore,carrick,elbaum@cse.unl.edu

ABSTRACT

Systems that interact with the physical world use software that represents and manipulates physical quantities. To operate correctly, these systems must obey the rules of how quantities with physical units can be combined, compared, and manipulated. Incorrectly manipulating physical quantities can cause faults that go undetected by the type system, likely manifesting later as incorrect behavior. Existing approaches for inconsistency detection require code annotation, physical unit libraries, or specialized programming languages. We introduce Phriky-Units[1], a static analysis tool that detects physical unit inconsistencies in robotic software without developer annotations. It does so by capitalizing on existing shared libraries that handle standardized physical units, common in the cyber-physical domain, to link class attributes of shared libraries to physical units. In this work, we describe how Phriky-Units works, provide details of the implementation, and explain how Phriky-Units can be used. Finally we present a summary of an empirical evaluation showing it has an 87% true positive rate for a class of inconsistencies we detect with high-confidence.

CCS CONCEPTS

•Software and its engineering → Software testing and debugging;

KEYWORDS

physical units; program analysis; static analysis; unit consistency; dimensional analysis; type checking; robotic systems

ACM Reference format:
John-Paul Ore, Carrick Detweiler, Sebastian Elbaum. 2017. Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 4 pages.
DOI: 10.1145/3092703.3098219

[1]Phrikê ("freaky") is the Greek spirit of horror (*φρικη*).

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3098219

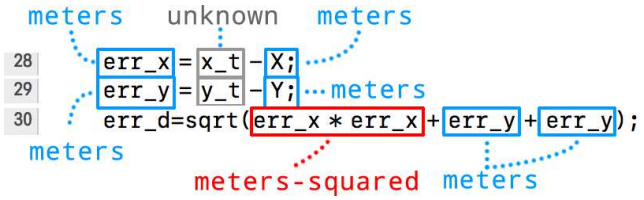

Figure 1: Example of code that incorrectly adds `meters-squared` to `meters` at line 30.

source: **https://git.io/vytcm**

1 INTRODUCTION

Software for systems that interact with the physical world manipulates quantities that have a physical meaning, like length, time, and velocity. One category of faults in robotic software is *dimensional and unit inconsistencies* [4], those associated with violations of the rules governing how physical quantities can be combined and manipulated while retaining their physical meaning. Figure 1 shows such a violation, incorrectly adding `meters` to `meters-squared`. The figure shows unit decorations to aid understanding. Although this code is syntactically correct and compiles, it hides a latent fault.

Developers can reduce the likelihood of physical unit inconsistencies by using annotations [15] or specialized languages with physical unit support built-in, like `F#` [8]. However, these approaches come with code migration or annotation costs, discouraging widespread use. Phriky-units is a tool that performs a lightweight static analysis to help developers detect these kinds of faults without requiring additional programmer annotations or code changes.

The high-level flow of Phriky-Units is shown in Figure. 2. As shown in the figure, Phriky-Units takes as input a target program and a mapping from attributes in shared libraries to units. Phriky-Units uses the mapping to decorate code with physical units and then propagates units in expressions. Phriky-Units uses the rules of dimensional analysis [4] to find inconsistencies. By design, the tool trades soundness and completeness for scalability and speed while still detecting meaningful inconsistencies, mirroring many popular static bug-finding tools [3].

The contributions of this work and tool are:

- Phriky-Units, a lightweight physical unit inconsistency detection tool for C++
- An instantiation of Phriky-Units for C++ files using the popular Robot Operating System[2] (ROS) [13].

[2]ROS is "maybe the most popular robotic middleware" [9], +3000 citations, +2500 systems, and nine million package downloads/month.

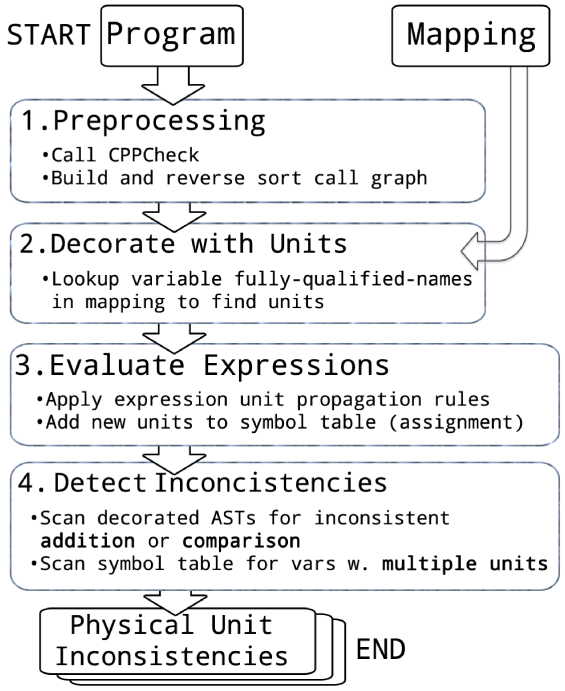

Figure 2: Approach Overview

- Source available at http://nimbus.unl.edu/tools.
- A summary of an empirical evaluation of Phriky-Units previously presented in [12].
- Examples of inconsistencies detected by Phriky-Units.

2 TOOL APPROACH

This section describes our approach to building a tool to detect physical unit inconsistencies. We first describe the goals and design considerations of the tool approach at a high level, then discuss the 'mapping' between attributes of shared libraries and units that is the source of unit information.

The goal of our approach is to provide lightweight detection of physical unit inconsistencies that is fast enough to be part of the build process while still detecting meaningful inconsistencies. During design, we explored tradeoffs between precision and feasibility and we accept design choices that compromise soundness and completeness to keep our approach lightweight and practical. Phriky-Units implements a static analysis that is semi-flow-sensitive (a simplified forward dataflow), path-insensitive, context-insensitive, and intra-procedural. For full details of our analysis please see [12]. We now discuss the mapping between attributes in shared libraries and units.

Mapping. Our approach requires a one-time effort *per domain* to create a mapping from attributes in shared libraries to physical units, so the whole domain of ROS programs can be checked as the result of this one-time effort. The mapping is a lookup table between attributes of shared libraries and physical units. Since many robot developers use shared libraries, the one-time effort to create the mapping enables physical unit inconsistency checking for all programs that use those libraries. This approach has larger efficiencies at larger scales. Unlike annotation, the mapping is separate from the code, offering several advantages to developers: 1) do not need to annotate their own code with physical units; 2) avoids the need for special annotated copies of shared libraries; and 3) unnecessary to have specialized build tools.

```
ros_unit_dictionary['nav_msgs::Odometry'] =
   {'position':     {'meter': 1.},
    'orientation':  {'quaternion':1.},
    'stamp':        {'second':1.},
    'linear':       {'meter':1.,'second':-1.},
    'angular':      {'second':-1.}}
ros_unit_dictionary['geometry_msgs::Pose'] =
    {'position':    {'meter': 1.},
    'orientation':  {'quaternion':1.}}
```

Figure 3: Mapping implementation as Python dictionary-of-dictionaries showing two entries. Each entry links an attribute from a shared library to units.

This mapping achieves the same effect as if all developers added unit annotations to their shared libraries.

Figure 3 shows the implementation of the mapping for two shared library structures in ROS. As shown in the figure, the mapping is a lookup table from Fully-Qualified Names (FQNs) to corresponding units.

Phriky-Units takes the mapping and a target program an input, as shown in Figure 2. Then Phriky-Units works in four phases as shown in the figure, and we now discuss each of the four phases.

Pre-processing. First, Phriky-Units pre-processes the file using CPPCheck [10] to create an XML representation of the AST and symbol table. Phriky-Units then constructs a context-insensitive call graph (without alias analysis) and sorts the call graph in reverse topological order to analyze procedures bottom-up. This yields a sorted list of procedures for Phriky-Units to analyze.

Decorate with Units. In the second phase Phriky-Units decorates variables with units. For each procedure, every statement is represented in the CPPCheck XML file as an Abstract Syntax Tree (AST). Phriky-Units implements a visitor pattern, and visits every node in a statement's AST, looking for variables. Phriky-Units associates variables with units in two ways: 1) by searching the symbol table for previously assigned units; and 2) with the mapping. Phriky-Units uses CPPCheck's symbol table to associate variables with FQNs, and the mapping associates FQNs to physical units, and shown in Figure reffig:mapping-dictionary. Once all variables have been decorated, Phriky-Units can examine and evaluate a statement's expressions.

Evaluate Expressions. In the third phase, Phriky-Units evaluates expressions in each statement's AST and propagates units according to unit propagation rules. For a full discussion of the unit propagation rules, please refer to [12]. For example, the code in Figure 1 on line 30 has been labeled with units for convenience and shows the `sqrt` and addition of several terms. During the previous decoration phase, each variable is labeled with meters based on knowledge of variables X and Y from another part of the program. During the evaluate expressions phase, the multiplication and addition operators are examined the resulting units are inferred. The $err\_x * err\_x$ yields `meters-squared` while the $err\_y + err\_y$ yields

meters. During expression evaluation Phriky-Units propagates the units to the assignment statement.

Detect Inconsistencies. The fourth, final phase detects three kinds of inconsistencies: 1) addition (and subtraction) of inconsistent units; 2) comparison of inconsistent units; and 3) assignment of multiple units. During this phase, Phriky-Units makes a final pass over the whole program, looking for statements with addition or comparison of inconsistent units. Finally, Phriky-Units performs a linear scan of the symbol table looking for variables that have been assigned multiple units. Note that multiplication by itself cannot be inconsistent since all unit combinations are allowed, but that the units resulting from a multiplication might cause an assignment of multiple units inconsistency.

For the inconsistency in Fig. 1, the error message is:

```
Addition of inconsistent units on line 30 with
high-confidence. Attempting to add ['meter': 2.0] to
['meter': 1.0]
```

3 IMPLEMENTATION DETAILS

Phriky-Units is a command-line tool implemented in 3300 lines of Python. It makes external calls to CPPCheck [10] to parse and preprocess source files, using default parameters: `cppcheck --dump -I ../include myfile.cpp`, which generates an XML 'dump' file using the default compiler directives. We imagine that Phriky-Units will be used in three ways: 1) to apply physical unit information to existing CPP files and access the physical-unit information at runtime, so software researchers can explore connections between software analysis and the physical quantities represented by program variables; 2) to check for physical unit inconsistencies either by robot software developers (who create new code) as well as software researchers looking to measure the frequency and kinds of these inconsistencies across corpora or in relation to other software quality metrics; 3) to teach about hazards specific to software representing physical quantities, especially by software educators teaching software for robotic systems.

To facility the first of these ways, accessing the physical unit assigned to variables at runtime, we structured Phriky-Units so that the main file `cps_units_checker.py` has a decoration phase and an inconsistency detection phase. Researchers looking to access the physical units at runtime can add their own code after Phriky-Units has decorated and propagated physical units in expressions. This new code might strengthen inference with new constraints or apply new detection techniques.

The second and the third ways to use Phriky-Units both involve simply using the tool at the command line with a target C++ file with the default parameters mentioned above.

Users of Phriky-Units can also explore the command line options that enable: 1) changing the confidence level for inconsistencies from high to low; 2) writing results to a SQL database; 3) printing program statements with the physical units assigned to each variable and the confidence of the unit assignment. Additionally, the command line options expose some experimental features: 1) using variable naming heuristics to decorate with units (i.e. *x_accel* for acceleration); 2) creating unit annotation 'placeholders' that can be used add units to variables and constants by hand; and 3) optionally applying these hand annotations.

Creating the mapping for ROS was an iterative process, aided by our familiarity with it. Overall, it took 3-4 days to build the mapping for 7 shared libraries, 82 classes, with 246 attributes representing 17 distinct physical units. Our mapping effort was aided by the fact that two of the authors are proficient users of ROS. We followed an iterative process, finding an individual attribute that represents quantities with physical units, and then examining the shared library containing that attribute and looking for other attributes with physical units, and then looking for more attributes in both the ROS documentation and ROS source code.

Extending Phriky-Units to other domains requires making a new mapping. The file `mapping.py` contains the encoding of the mapping. As shown in Figure 3, we encode units as a python dictionary and use strings like `'meters'` for readability. The figure shows an example of the FQN to unit mapping. The first entry is for the shared library `nav_msgs` containing the structure `odometry` with five attributes that have physical units. Attributes without units are omitted.

During the design and implementation of Phriky-Units, we made several decisions affecting the trade-off between precision, recall, speed, and scalability. Since this is an initial version, we steered toward a tool that could demonstrate the potential of this approach without including all possible features. Specifically, we chose a flow-insensitive analysis to enable a quick, linear scan, and we reason with incomplete information (where incomplete information results in 'low-confidence' unit assignments and inconsistencies) to explore the space. Although these decisions compromise soundness and completeness, many other useful static analysis tools make this compromise as well [3], and we believe Phriky-Units can be useful in the burgeoning robotic software development community.

Phriky-Units can be installed with `pip install phriky_units` (tested on Ubuntu 16.04 and OSX 10.12.4), and requires CPPCheck 1.75 or higher. For additional details including source, visit http://nimbus.unl.edu/tools/.

4 EXAMPLES AND EVALUATION SUMMARY

In this section we provide examples of the kinds of inconsistencies Phriky-Units can detect. We also summarize our empirical evaluation previously published in [12]. Note that our previous work also provides measurements of Phriky-Units' speed.

Figure 4 shows an example of an inconsistent comparison, where on line 20 the variable `max_vx` receives the units `meters` by assignment. The `meters` source is from the attribute *x* from the structure `Point` in the shared library `geometry_msgs`. Then on line 53, the code compares `max_vx` with `msg.linear.x`, which has units `meters-per-second`. This comparison is inconsistent because these variables do not have the same physical units, as required by dimensional analysis.

Figure 5 shows an assignment of multiple units inconsistency arising when a variable with `meters-per-second` is assigned to a variable that already has different units, `per-second` (angular velocity). Both of these variables are decorated with units because they are attributes of the structure `Twist::linear` in the shared library `geometry_msgs`. This is likely a simple copy-and-paste error. However, without specialized unit inconsistency detection, this code compiles and might be only revealed by incorrect behavior of the system in the physical world.

```
19 void limits_Callback(const geometry_msgs::Point& msg){
20     max_vx = msg.x;
                meters
53 vx = msg.linear.x > max_vx ? max_vx : msg.linear.x;
meters-per-second
```

Figure 4: Example of inconsistent comparison found by Phriky-Units that incorrectly compares `meters-per-second` to `meters` on line 53.

source: https://git.io/v9RRM

```
264 void getAttackerPose(const geometry_msgs::Twist& msg) {
265     attacker_position.linear.x = msg.linear.x;
266     attacker_position.linear.y = msg.linear.y;
267     attacker_position.angular.z = msg.linear.z;
                per-second              meters-per-second
```

Figure 5: Example of multiple units inconsistency found by Phriky-Units that incorrectly assigns `meters-per-second` to `per-second` on line 267.

Summary of Effectiveness. As reported previously [12], we evaluated Phriky-Units on a corpus of 213 open-source robotics projects, containing 934, 124 non-blank non-commented lines of C/C++ as reported by CLOC [6]. The projects in the corpus are listed at http://nimbus.unl.edu/tools. We found 217 physical unit inconsistencies in 11% of the 213 projects, and manually labeled them as True Positive (TP) or False Positive (FP) for each inconsistency. We calculated precision $TP\% = 100 * TP/(TP + FP)$ and found an 87% TP rate for inconsistencies we can detect with high confidence, including 'Addition of Inconsistent Units' (Figure 1), 'Comparison of Inconsistent Units' and 'Assignment of Multiple Units.' We do not calculate recall because the true number of False Negatives in this corpus is unknown.

5 RELATED WORK

Supporting physical units through type checking has been studied since at least the late 1970s [7], and language or compiler support has been proposed for C [17], Java [18], and most recently realized in F# [8]. These approaches require either annotations or specialized languages. Roşu's and Feng proposed dynamic physical unit checking in C [14]. Static support for C++ through templates is built into Schabel and Watanabe's `boost::units` [15] for C++. Unlike these approaches, Phriky-Units requires neither programmer annotations nor compiler extensions, and works with existing code without modification.

We are not aware of an authoritative study of the frequency or severity of these kinds of unit inconsistencies, but there are instances of spectacular failures like the crash of the 1998 Mars Climate Orbiter [16].

6 CONCLUSION AND FUTURE WORK

The next steps for Phriky-Units focus on strengthening the analysis while maintaining speed and scalability. We intend to construct mappings for other robotic middleware like Orocos [5], OpenRTM [1], MOOS [2], and Yarp [11]. We would like to explore more sophisticated and robust analysis frameworks like Clang, to increase precision. Variable names are a potential source of units information because developers reuse idiomatic variables names for position, velocity, or duration (i.e. `x_pos`, `y_vel`, `d_t`). We intend to modify Phriky-Units to measure the percentage of variables that are decorated with units by automatic inference, and how many remain unannotated. Since no fully-labeled datasets of unit inconsistencies are presently available, we would like to exhaustively label a small dataset identifying all true positives to determine the recall. We also intend to analyze successive versions of systems to see explore if unit inconsistencies are introduced and later corrected, and use this to inform repair recommendation.

This paper introduces Phriky-Units, a lightweight static analysis tool to detect physical unit inconsistencies in software systems that interact with the physical world. We assume shared libraries with attributes representing physical quantities. Phriky-Units has an 87% true positive rate for a class of inconsistencies we detect with high-confidence, and is available at http://nimbus.unl.edu/tools.

ACKNOWLEDGMENTS

This work was supported in part by NSF awards #1638099, #1526253, and #1526652, USDA-NIFA #2013-67021-20947, and USDA-NIFA #2017-67021-25924

REFERENCES

[1] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. 2008. A software platform for component based Rt-system development: OpenRTM-AIST. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 87–98.
[2] Michael R Benjamin, Henrik Schmidt, Paul M Newman, and John J Leonard. 2010. Nested autonomy for unmanned marine vehicles with MOOS-IvP. *Journal of Field Robotics* 27, 6 (2010), 834–875.
[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
[4] Percy Williams Bridgman. 1922. *Dimensional Analysis*. Yale University Press.
[5] Herman Bruyninckx. 2001. Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, Vol. 3. IEEE, 2523–2528.
[6] Al Danial. 2016. Count Lines Of Code. (2016). https://github.com/AlDanial/cloc
[7] Michael Karr and David B Loveman III. 1978. Incorporation of units into programming languages. *Commun. ACM* 21, 5 (1978), 385–391.
[8] Andrew Kennedy. 2008. Types for units-of-measure in F#: invited talk. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM, 1–2.
[9] Gergely Magyar, Peter Sinčák, and Zoltán Krizsán. 2015. Comparison study of robotic middleware for robotic applications. In *Emergent Trends in Robotics and Intelligent Systems*. Springer, 121–128.
[10] Daniel Marjamäki. 2013. Cppcheck: a tool for static C/C++ code analysis. (2013).
[11] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. 2006. Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems* 3, 1 (2006), 8.
[12] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Lightweight Detection of Physical Unit Inconsistencies without Program Annotations. In *Proceedings of the 2017 International Symposium on Software Testing and Analysis*. ACM. Accepted, to Appear.
[13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3.2. Kobe, Japan, 5.
[14] Grigore Rosu and Feng Chen. 2003. Certifying measurement unit safety policy. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 304–309.
[15] Matthias Christian Schabel and Steven Watanabe. 2008. Boost. *Units* 1, 0 (2008), 2003–2010.
[16] Arthur G Stephenson, Daniel R Mulville, Frank H Bauer, Greg A Dukeman, Peter Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. 1999. Mars climate orbiter mishap investigation board phase I report, 44 pp. *NASA, Washington, DC* (1999).
[17] Zerksis D Umrigar. 1994. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices* 29, 9 (1994), 135–139.
[18] André Van Delft. 1999. A Java extension with support for dimensions. *Software Prac. Experience* 29, 7 (1999), 605–616.