



Phys: Probabilistic Physical Unit Assignment and Inconsistency Detection

Sayali Kate
Purdue University, USA
skate@cs.purdue.edu

John-Paul Ore
University of Nebraska–Lincoln, USA
jore@cse.unl.edu

Xiangyu Zhang
Purdue University, USA
xyzhang@cs.purdue.edu

Sebastian Elbaum
University of Nebraska–Lincoln, USA
elbaum@cse.unl.edu

Zhaogui Xu
Nanjing University, China
zhaogui.xu@outlook.com

ABSTRACT

Program variables used in robotic and cyber-physical systems often have implicit physical units that cannot be determined from their variable types. Inferring an abstract physical unit type for variables and checking their physical unit type consistency is of particular importance for validating the correctness of such systems. For instance, a variable with the unit of ‘meter’ should not be assigned to another variable with the unit of ‘degree-per-second’. Existing solutions have various limitations such as requiring developers to annotate variables with physical units and only handling variables that are directly or transitively used in popular robotic libraries with known physical unit information. We observe that there are a lot of physical unit hints in these softwares such as variable names and specific forms of expressions. These hints have uncertainty as developers may not respect conventions. We propose to model them with probability distributions and conduct probabilistic inference. At the end, our technique produces a unit distribution for each variable. Unit inconsistencies can then be detected using the highly probable unit assignments. Experimental results on 30 programs show that our technique can infer units for 159.3% more variables compared to the state-of-the-art with more than 88.7% true positives, and inconsistencies detection on 90 programs shows that our technique reports 103.3% more inconsistencies with 85.3% true positives.

CCS CONCEPTS

• **Software and its engineering** → **Abstract data types**; **Software defect analysis**; • **Mathematics of computing** → **Factor graphs**;

KEYWORDS

abstract type inference; physical units; static analysis; unit consistency; dimensional analysis; probabilistic inference; robotic systems

ACM Reference Format:

Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. 2018. Phys: Probabilistic Physical Unit Assignment and Inconsistency Detection. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’18)*, November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3236024.3236035>

1 INTRODUCTION

Program variables representing physical units like meter or radian are common in robotic and cyber-physical systems. However, the types of these variables (e.g., float and double) can hardly denote such physical information. While compilers and many analysis techniques ensure variables are manipulated according to the typing rules, ensuring variables with physical units are manipulated according to the semantics of the physical world, however, is less common and yet as crucial for these kinds of systems. For example, a recent study found hundreds of faulty manipulations in robots using the ROS middleware [22]. Those systems built correctly but presented inconsistent unit manipulations such as assigning linear (meter-per-second) and angular (radian-per-second) units to a variable, or adding variables representing velocity (meter-per-second) and distance (meter).

Automated approaches to aid in the detection of inconsistent usage of variables representing physical units include unit-aware programming languages [1, 31], unit-aware libraries [9, 30], and unit type annotations [10]. These approaches, however, have not been broadly adopted in part because of their associated cost in modifying existing systems or changing entrenched development practices. Approaches that require no additional development investment are desirable but rare. One of such approaches, *Unify*, can detect unit usage discrepancies across versions [7], but it cannot detect unit inconsistencies when a variable is first introduced. Another approach is taken by *Ayudante* [8], which infers abstract type inconsistencies by contrasting clusters of variables based on dataflow versus clusters based on the meaning of variable names as per a large lexical database [18]. This approach, however, misses much of the unique constructive semantics of physical units (like $\text{meter}^2 = \text{meter} * \text{meter}$) and assumes that all these variable and name associations are certain, when in practice they are probabilistic. A more broadly applicable approach that requires no additional developer investment is *Phriky-Units (Phriky)*, which relies on one-time mapping of physical attributes in shared-libraries to units [23] to infer the units of variables, and uses a lightweight dataflow analysis and unit propagation to facilitate inconsistency detection using

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE ’18, November 4–9, 2018, Lake Buena Vista, FL, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5573-5/18/11...\$15.00

<https://doi.org/10.1145/3236024.3236035>

dimensional analysis. The one-time mapping and lightweight analysis that make *Phriky* cost-effective at detecting inconsistencies, also limit its power. In particular, we noticed that *Phriky* only assigns units to a small fraction of the variables of non shared-library data types that hold physical units (in this paper we quantify that space to be under 31.56%).

To address that limitation, we propose an approach that can tap on new sources of information to assign units to a larger portion of the variable space, facilitating the detection of more inconsistencies. The approach builds on two key insights. First, variables representing physical entities are often named and operated on to reflect those entities, giving hints about variables' units. For example, in the statement: $v = \text{angSpeed} * \text{wheel_diam}/2$, names '*angSpeed*' and '*wheel_diam*' suggest that they represent angular speed and length respectively, and a multiplication operation on them indicates that variable v is intended to represent linear velocity. Second, since the correctness of these hints is not certain (developers can violate naming conventions or inappropriately operate on units) we must deal with them probabilistically. For example, since name '*wheel_diam*' does not use the entity term '*diameter*' completely, we can only say that it is likely to have unit 'meter' with some probability.

Put together, these insights indicate that there are hints to be leveraged but to do so we need to model the sources of uncertainty in terms of probabilities. More specifically, our approach: 1) collects initial observations (or *beliefs*) based on various evidences such as variable names and expression forms that suggest physical unit (e.g., expression $x > \pi$ indicates x has unit 'radian'), and encodes them as initial probabilities to model uncertainty; 2) analyzes the code to generate five kinds of probability constraints that denote dependencies between variable units; 3) generates a graph where the nodes are the initial observations and constraints transformed into functions on the variables, and the edges connect the functions with their corresponding variables; and 4) performs belief propagation [25] along the edges to determine the posterior probabilities denoting the likelihood that a variable would have an associated physical unit. Once the variables have a physical unit assigned, detection of inconsistency is performed following established dimensional analysis rules [3].

The contributions of our work are:

- A probabilistic approach for physical unit inference and inconsistency detection that takes advantage of variables' names, expression forms, and associated operations to make probabilistic inferences of unit types.
- A prototype of the approach implemented in a tool, *Phys*, that assigns units and detects inconsistencies on C++ programs.
- An evaluation on 90 sample ROS-based project files. The assessment shows that *Phys* can infer units for 159.27% more variables of non shared-library data types in 30 sample files (with more than 88.7% true positives), and detect 103.31% more inconsistencies in 90 sample files (with 85.3% true positives), when compared with the state-of-the-art.

2 BACKGROUND

In this section, we introduce the basic notations of physical units and unit-inconsistencies, and provide a brief overview of probabilistic inference based on graphical models.

2.1 Physical-units and Unit-inconsistency Detection

In robotic software, some variables represent the physical dimensions such as length, velocity, and acceleration. Each of these variables carries a physical-unit, e.g., a variable representing length stores a value of unit 'meter'. Operations on such variables need to follow certain dimensional rules, e.g., a length value cannot be added to a velocity value. Violation of such a rule, we refer to as *unit-inconsistency*. In order to detect a unit-inconsistency, we first need to collect the physical-unit information of each variable. However, physical-unit information is not explicitly declared for variables unlike type information. For example, while length and velocity carry different physical-units, 'meter' and 'meter-per-second', respectively, they may be represented by variables of the same type, e.g., 'float'. Therefore, traditional type checking of a software program cannot detect unit-inconsistencies. A new kind of analysis is required to detect unit-inconsistencies. These analyses focus on inferring the physical-units of variables. There have been a number of previous works that aim to address this challenge such as the tools *Phriky* [23, 24] and *Osprey* [10].

In particular, *Phriky* performs unit consistency analysis on programs that use a robotic shared-library containing data-types for various physical quantities. These shared-library data-types provide the basic unit information for a subset of variables as a starting point for the analysis. Such initial unit information is propagated to other variables through a set of inference rules similar to typing rules. Specifically, the tool *Phriky* implements a lookup table, called a 'mapping,' from attributes of shared libraries to physical units for software written for the Robot Operating System (ROS). ROS is a publisher-subscriber middleware that defines commonly used messages in shared libraries. These shared messages have attributes with physical meanings like lengths, velocities, and accelerations.

Our tool *Phys* leverages this one-time 'mapping' during analysis as one way to find variables with units. Also, it uses the same notation for physical-units and unit-inconsistencies as defined by *Phriky*. The notation is described below.

Physical-units. The physical-unit representation contains a standard set of units from the specification of International System of Units (SI) [2], plus some units officially accepted to be used with the SI system. Following [10], units are defined as:

$$u := \text{meter} \mid \text{second} \mid \text{kilogram} \mid \text{quaternion} \mid \text{radian} \mid \text{degree_360} \mid \text{amp} \mid \text{candela} \mid \text{degree_celsius} \mid \text{unknown} \mid \text{dimensionless} \mid u_1 * u_2 \mid u^{-1} \quad (1)$$

Unit *unknown* means that the unit of a variable is not known. The unit *dimensionless* means a variable does not have a unit, such as a scaling factor or the ratio 'meter-per-meter'. The product $u_1 * u_2$ represents a multiplication of two units and u^{-1} represents the inverse of a unit. Together, product and inverse form various derived units like 'meter * second⁻¹', i.e., a unit of velocity, 'meter-per-second'. Further, note that we use the same unit to represent variables of the same dimension. So, two variables of the length dimension with different units in practice, 'centimeter' and 'meter' respectively, are assumed to have the same unit, 'meter'.

Unit-inconsistency Detection. The violation of dimensional rules, such as one can only add or compare values of the same dimension, are translated to unit-inconsistencies over program constructs. They are listed below. Let u_1 and u_2 be two different units.

(1) Addition/subtraction of inconsistent units: an inconsistency is detected when there is an addition/subtraction of two different units. Note that multiplication and division of inconsistent units may be legitimate, such as ‘meter-per-second * second’, and hence not a good standard for inconsistency detection [23].

$$u_1 (+, -) u_2$$

(2) Comparison of inconsistent units: an inconsistency is detected when two different units are compared to each other.

$$u_1 (<, \leq, =, \neq, \geq, >) u_2$$

(3) Assignment of inconsistent units. This category includes two cases: a) the left-side and the right-side of an assignment have different units; b) the right-side of an assignment has two different units, e.g., a right-side variable may have different units in the two branches of a conditional statement. Note that we union the units.

$$u_1 \leftarrow u_2, x \leftarrow \{u_1, u_2\}$$

(4) Function with different unit arguments: an inconsistency is detected when a function’s i^{th} argument receives values with different units in two different function calls.

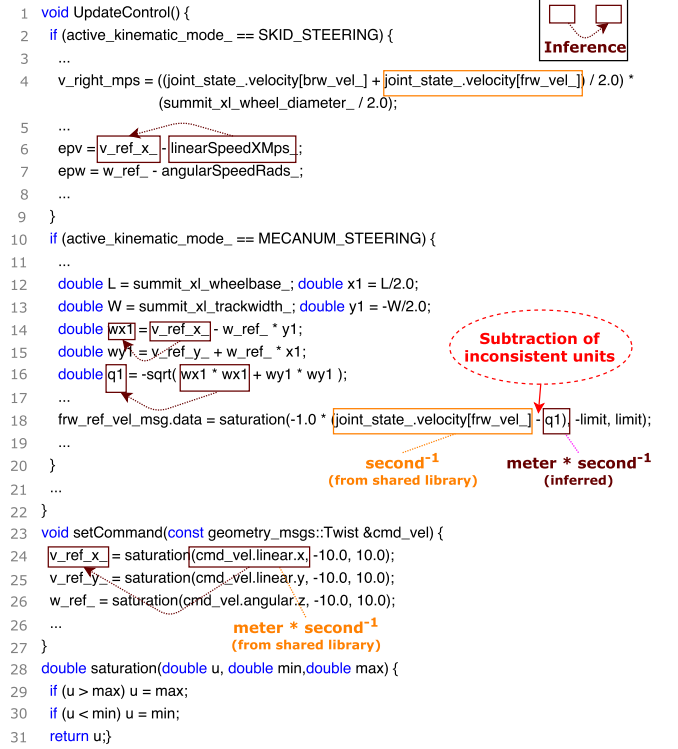
$$f(u_1), f(u_2)$$

Detection of unit-inconsistencies serves two purposes: 1) the inconsistent use of units may be intentional as per the developer. In such a case, it is always recommended to document a suspicious use of units, especially when the code is meant to be reused. This makes it easier to maintain the code; 2) the inconsistent use of units exposes the source of a potential unintended system behavior, or a bug.

2.2 Probabilistic Inference with Graphical Models

Often while solving real-world problems, we need to draw conclusions based on incomplete or uncertain information. Uncertainty is usually modeled in the form of a probability distribution. The process of performing inference based on such models is called *probabilistic inference*. One type of probabilistic inference is the computation of a marginal probability of the event or property. For example, in our case, we need to compute the marginal probability that a variable x has unit u from the probability distribution of all variables in the program being of unit u . The probability that x has unit u is conditioned on how x is used in the program and depends on what is assigned to x and how x is used in mathematical expressions. Factor graphs [14] are used to represent the structure of this conditional dependence.

In factor graphs, a random variable (or a boolean variable with probability, “ r is 0.7 chance true”) is used to denote a predicate (e.g., program variable x has unit u , or $P(x, u)$). Inter-dependent random variables are denoted as a propositional logic formula with probabilities. For instance, the information that a variable vel likely has the unit of ‘meter-per-second’ with probability 0.7 is denoted by $N(‘vel’, \text{meter-per-second}) \xrightarrow{0.7} P(vel, \text{meter-per-second})$. Intuitively, it means from the naming convention N (think of it as a dictionary that maps a name to its unit) we know that a name ‘ vel ’ has the ‘meter-per-second’ unit, we then have 0.7 confidence that variable vel is really of that unit, with probability 0.7 modeling the uncertainty of naming conventions (i.e., programmers may not

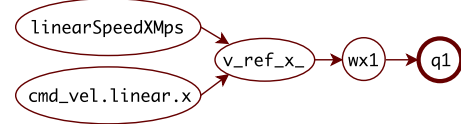


```

1 void UpdateControl() {
2   if (active_kinematic_mode_ == SKID_STEERING) {
3     ...
4     v_right_rmps = ((joint_state_.velocity[brw_vel_] + joint_state_.velocity[frw_vel_] / 2.0) *
5                     (summit_xl_wheel_diameter_ / 2.0));
6     ...
7     epv = v_ref_x_ - linearSpeedXMps;
8     epw = w_ref_ - angularSpeedRads;
9     ...
10  }
11  if (active_kinematic_mode_ == MECANUM_STEERING) {
12    ...
13    double L = summit_xl_wheelbase_; double x1 = L/2.0;
14    double W = summit_xl_trackwidth_; double y1 = -W/2.0;
15    double wx1 = v_ref_x_ - w_ref_ * y1;
16    double wy1 = v_ref_y_ + w_ref_ * x1;
17    double q1 = -sqrt(wx1 * wx1 + wy1 * wy1);
18    frw_vel_ref_msg.data = saturation(-1.0 * (joint_state_.velocity[frw_vel_] - q1) - limit, limit);
19    ...
20  }
21  ...
22  void setCommand(const geometry_msgs::Twist &cmd_vel) {
23    v_ref_x_ = saturation(cmd_vel.linear.x, -10.0, 10.0);
24    v_ref_y_ = saturation(cmd_vel.linear.y, -10.0, 10.0);
25    w_ref_ = saturation(cmd_vel.angular.z, -10.0, 10.0);
26    ...
27  }
28  double saturation(double u, double min, double max) {
29    if (u > max) u = max;
30    if (u < min) u = min;
31    return u;
32  }

```

(a) Code snippet showing several constraints related to variable $q1$. source: <https://git.io/vAAAI>



(b) Inference resulting in meter-per-second units for $q1$.

Figure 1: Example unit inconsistency detected using probabilistic constraints.

respect naming conventions). A random variable may be involved in multiple propositional logic formulas denoting its dependencies. For instance, if the program has an assignment statement $x = y$, we have $P(y, U) \xrightarrow{0.95} P(x, U)$. The factor graph engine will take these formulas, derive the corresponding joint probability distribution, and perform probabilistic inference. There are various inference algorithms, both exact and approximate, defined for these graphical models. The approximate algorithms allow us to find solutions where the exact inference is infeasible. After inference, the post-distribution denotes the fusion of all the (uncertain) evidences and hence our analysis results. A detailed description of probabilistic graphical models representation and inference can be found in [13].

Phys uses a factor graph model to represent the joint probability distribution of variables being of unit u . And, since there can be a large number of variables in the program, it uses an approximate algorithm for the unit inference.

3 MOTIVATING EXAMPLE

Figure 1a shows a code-snippet from a ROS-based project file available on GitHub, ‘summit_xl_robot_control.cpp’. *Phys* reports

an inconsistency on line 18. The nature of the inconsistency is that the robot's low-level wheel controllers are commanded an incorrect reference velocity. This inconsistency is difficult to detect statically or at run-time because it is syntactically and semantically correct as per the programming language, and the wheel turns in the correct direction but the velocity is incorrect in proportion to the radius of the robot's wheel (it is not apparent for smaller wheels). As a result, under some scenarios, the robot can move undesirably slow. In such cases, control engineers often blame the low-level controller and try to compensate by tuning control gains, i.e., changing parameters to make the motors more sensitive to the (faulty) signal, leading to potential instabilities.

We are motivated by these kinds of stealthy, difficult-to-detect bugs that require combining multiple information sources each with a different degree of certainty—from less certain variable name hints to more certain dataflow hints.

Going back to our example in Figure 1a, to detect the 'subtraction of inconsistent units' in line 18, we need to infer units of `joint_state_.velocity` and `q1` (`frw_vel_` omitted for brevity). Inferring units for variables like `joint_state_.velocity`, which instantiate shared ROS libraries attributes with known unit types, is already done successfully by *Phriky* using predefined maps so we reuse that approach. More specifically, variable `joint_state_.velocity` is an attribute of class `sensor_msgs::JointState`, and the mapping determines that the attribute `JointState::velocity` have units 'per-second'.

Variable `q1`, however, does not instantiate anything with a known unit. Inferring units for such variables requires a new and more sophisticated approach for unit inference, as used by our tool *Phys*. We note that variable `q1` on line 16 is assigned units computed from the units of `wx1` (and `wy1`; description for `wy1` omitted for simplicity). `wx1` is assigned units on line 14, providing a dataflow constraint from `v_ref_x` because `v_ref_x` is part of an addition, and we assume all units within an addition are the same. For inferring the units of `v_ref_x` we have two hints: 1) a dataflow constraint on line 6 from `linearSpeedXMps_` along with a naming hint because `linearSpeedXMps_` contains the substring 'Speed'; and 2) assignment of units resulting from the procedure `saturation` on line 24. Interestingly, `saturation` uses its first parameter as the return variable and thus provides a hint that its returned units may be the same as its first argument's units. Therefore, on line 24, `v_ref_x` is likely to have the same units as argument `cmd_vel.linear.x` that has a known unit associated as part of the map.

All these hints together form a set of probabilistic constraints, partially shown in Figure 1b as an inference graph. *Phys* then calculates the likelihood of possible unit assignments for `q1` and assigns the most likely unit, which turns out to be 'meter-per-second' with the highest probability of 0.81, reporting that:

Addition of inconsistent units on line 18.
Attempting to add [{'second': -1.0}] to
[{'second': -1.0, 'meter': 1.0}].

4 APPROACH

In this section, we first provide a high-level overview of *Phys*, then give a detailed description of the probabilistic constraints generated by *Phys*, and finally discuss how the probabilistic inference engine transforms probabilistic constraints into a factor graph and conducts graph inference.

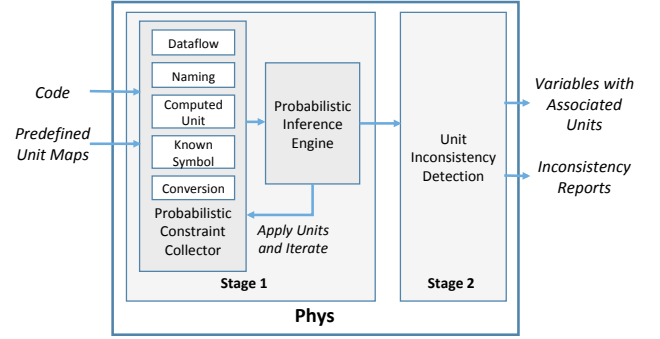


Figure 2: Overview of Phys framework.

Table 1: Constraint Predicate Definitions

TYPE	SYMBOL	DEFINITION
Dataflow	$D(var, u)$	var has unit u based-on dataflow dependency of var on ROS-TYPED variable.
Naming	$N(var, u)$	var has unit u based-on its name.
Computed-Unit	$C(var, u)$	unit u of var is computed from the right side of assignment statement, ' $var = expr$ '.
Known-Symbol	$K(var, u)$	var has unit u based-on known symbols.
Conversion	$F(var, u)$	var has unit u based-on unit-conversion expression.
Prior-Probability	$pred = 1(q)$	$pred$ is true with probability q .
Implication	$pred1 \xrightarrow{p} pred2$	$pred1$ implies $pred2$ with probability p .

Table 2: Collected constraints for example in Figure 1.

	LN#	Probabilistic Constraint	Iter#
(1)	24	$C(v_ref_x, ms^{-1}) \xrightarrow{0.95} P(v_ref_x, ms^{-1})$	1
(2)	25	$C(v_ref_y, ms^{-1}) \xrightarrow{0.95} P(v_ref_y, ms^{-1})$	1
(3)	6	$P(v_ref_x, ms^{-1}) \xleftrightarrow{0.95} P(linearSpeedXMps, ms^{-1})$	1
(4)	-	$N(linearSpeedXMps, ms^{-1}) \xrightarrow{0.7} P(linearSpeedXMps, ms^{-1})$	1
(5)	14	$P(wx1, ms^{-1}) \xleftrightarrow{0.95} P(v_ref_x, ms^{-1})$	1
(6)	15	$P(wy1, ms^{-1}) \xleftrightarrow{0.95} P(v_ref_y, ms^{-1})$	1
(7)	16	$C(q1, ms^{-1}) \xrightarrow{0.95} P(q1, ms^{-1})$	2
(8)	-	$N(summit_xl_wheelbase, m) \xrightarrow{0.7} P(summit_xl_wheelbase, m)$	1
(9)	-	$N(summit_xl_trackwidth, m) \xrightarrow{0.7} P(summit_xl_trackwidth, m)$	1
(10)	12	$P(L, m) \xleftrightarrow{0.95} P(summit_xl_wheelbase, m)$	1
(11)	13	$P(W, m) \xleftrightarrow{0.95} P(summit_xl_trackwidth, m)$	1
(12)	26	$C(w_ref, s^{-1}) \xrightarrow{0.95} P(w_ref, s^{-1})$	1
(13)	7	$P(w_ref, s^{-1}) \xleftrightarrow{0.95} P(angularSpeedRads, s^{-1})$	1
(14)	-	$N(angularSpeedRads, s^{-1}) \xrightarrow{0.7} P(angularSpeedRads, s^{-1})$	1
(15)	12	$C(x1, m) \xrightarrow{0.95} P(x1, m)$	2
(16)	13	$C(y1, m) \xrightarrow{0.95} P(y1, m)$	2
(17)	14	$C(wx1, ms^{-1}) \xrightarrow{0.95} P(wx1, ms^{-1})$	3
(18)	15	$C(wy1, ms^{-1}) \xrightarrow{0.95} P(wy1, ms^{-1})$	3

Figure 2 shows an overview of the *Phys* framework. It is divided into two stages: Stage 1 infers units with the help of a probabilistic inference engine, and Stage 2 uses the inferred units to detect unit inconsistencies.

Stage 1: Probabilistic Unit Inference. There are two main components for unit inference as shown in Figure 2 Stage 1: probabilistic constraint collector and probabilistic inference engine. The constraint collector first preprocesses the code to generate a list of functions and then scans each function to identify variables that instantiate ROS shared library data types. Using a predefined map

from ROS attributes to units [24], these variables can be directly assigned a physical unit. The mapping is a one-time effort and consists of 98 data structures, each having 2-5 fields. For example, variables instantiating the ROS attribute *geometry_msgs::Twist.linear.x* are mapped to ‘meter-per-second’. The mapping provides a subset of variables with known units and allows us to transfer this known unit information to variables of non-shared library type and then fuse it with other unit hints.

Next, the constraint collector traverses through the code to gather unit hints called *observations* and to derive constraints that denote relations between variables. Table 1 defines five types of observations (dataflow, naming, computed-unit, known-symbol, and conversion) denoted as predicates asserting a variable *var* has a unit *u*. These observations are associated with some prior probabilities to express the initial confidence in those observations to be true, which is captured by the constraint type defined in the sixth row of Table 1. The collector also constructs implication constraints, as per the seventh row of Table 1, which capture the inter-dependences of the predicates based on program semantics, allowing probabilities to be propagated and fused. Last, the constraint collector records composite units such as ‘meter-per-second-squared’, the result of combining units in mathematical expressions. The observed and composite units are added to the set *UNIT_SET*.

After all constraints have been collected, the probabilistic inference engine transforms the constraints into a factor graph. The engine performs belief propagation in the graph for each unit in *UNIT_SET*. This yields a posterior marginal probability for each variable, *var*, having a unit, *u*, denoted as $P(\text{var}, u)$. If there is some evidence for *u* (i.e., with probability $p > 0.5$, where 0.5 is no knowledge), and *u* is more likely than any other unit, then *var* is assigned the unit *u*. As shown in Figure 2, Stage 1 includes an iterative process of gathering constraints, running the probabilistic inference engine, inferring units, and again gathering constraints. The iterative part is repeated until a fixed point is reached.

Stage 2: Unit-inconsistency Detection. The unit inconsistency detector scans the annotated abstract syntax tree (AST) for unit inconsistencies as defined in Section 2.1. That is, inconsistent addition/subtraction, comparison, assignment, or function arguments. To mitigate false positives, the detector is conservatively configured by default to report an inconsistency only if the three most likely unit assignments to the variables involved in an expression all yield an inconsistency. *Phys* then emits a list of variable unit assignment and any detected inconsistencies.

4.1 Probabilistic Constraints

Phys has two forms of constraints: the prior-probability constraints, and the implication constraints.

As shown in Table 1, a prior-probability constraint encodes that an initial observation *pred* is true with some confidence *q*, and is denoted $\text{pred} = 1(q)$. This constraint encodes belief from prior human domain knowledge and distributions of known types inferred solely from ROS libraries. The inherent uncertainty in this constraint can be substantially suppressed when the inference engine fuses information from many sources.

Implication constraints are used to relate two predicates/random-variables together, and take the form $\text{pred1} \xrightarrow{p} \text{pred2}$ with confidence *p*, and can also be bidirectional. Table 2 shows implication

constraints collected from the code snippet in Figure 1a. Notice how every constraint in the table is formulated as an implication constraint, from an observation predicate to a posterior predicate such as (1) and (2), or from a posterior predicate to another posterior predicate such as (10) and (11). Next, we discuss how to collect the constraints from the data flow, naming, computed-unit, known-symbol and conversion perspectives.

Dataflow Constraints. Dataflow constraints are collected for variable pairs that can have the same unit due to program dataflow. For example, Figure 1a has a dataflow constraint on line 14: *wx1* and *v_ref_x*, and the generated constraint is shown in Table 2 (5). In this example, the dataflow constraint encodes the dimensional rule that the units resulting from addition/subtraction are likely the same as the units of the operands. As shown in the table, this constraint has a probability of 0.95. It is a standard to use 0.95 to represent high confidence in the inference [32].

More generally, various program expressions such as addition, comparison, *min()*, *max()* function calls, and copy provide unit hints about their operands according to the dimensional rules. The operands of such expressions or statements potentially represent quantities with the same unit. If a dataflow relation for the same unit is detected for variables *a* and *b*, we add an implication constraint between the two predicates: $P(a, u) \xrightarrow{0.95} P(b, u)$, where $u \in \text{UNIT_SET}$ and confidence is propagated in both directions.

For variables with unit hints from the ROS mapping, we formulate the following constraints: 1) a prior probability constraint, $D(b, K) = 1(0.95)$ with *D* asserting the unit of a ROS variable; and 2) an implication constraint, $D(b, K) \xrightarrow{0.95} P(b, K)$.

Naming Constraints. Developers tend to use variable names that hint at the physical quantities they represent. For example, *linearSpeedXMps_* contains ‘Speed’ that suggests a linear velocity. *Phys* uses a hand-coded lookup table between common strings (called ‘suffixes’) and units. For example, ‘length’ and ‘distance’ are mapped to ‘meter’. Generating the table is a one-time effort. The current version contains only 41 entries.

To find the best suffix match, *Phys* uses a similarity metric:

$$\text{sim}(\text{var}, u) = \max_{s \in \{(s:u)\}} \frac{\text{len}(\text{LCS}(\text{var}, s, k))}{\text{MAX_LEN_SUFFIX}} \quad (2)$$

Here, $\text{sim}(\text{var}, u)$ computes the maximum similarity between variable *var* over all suffixes *s*, where $\{(s:u)\}$ is the set of all suffixes with the same unit *u*. The term $\text{len}(\text{LCS}(\text{var}, s, k))$ represents the length of a longest common substring between a variable *var* and a suffix *s* such that the length is at least *k* and the substring starts with the first *k* characters of a suffix *s* ($k = 3$ in our implementation). The longest hand-coded suffix, $\text{MAX_LEN_SUFFIX} = 12$.

The maximum similarity score $\text{sim}(\text{var}, u)$ is then converted to a naming constraint:

$$N(\text{var}, u) = 1(p = 0.5 + 0.5 * \text{sim}(\text{var}, u)) \quad (3)$$

The confidence *p* is scaled so that a similarity of 0 is a confidence of 0.5, meaning ‘no evidence’. An implication constraint is also generated. For *linearSpeedXMps_*, it is:

$$N(\text{linearSpeedXMps_ms}^{-1}) \xrightarrow{0.7} P(\text{linearSpeedXMps_ms}^{-1}) \quad (4)$$

The predicate $N(\text{linearSpeedXMps_ms}^{-1})$ has an initial confidence $0.5 + 0.5 * \text{sim}(\text{var}, \text{ms}^{-1})$ that is propagated to $P(\text{linearSpeedXMps_ms}^{-1})$ with probability 0.7.

Table 3: Probabilistic constraints.

Type	Expression	Condition	Probabilistic Constraints
Dataflow	$a \text{ op}_1 b, c ? a : b$ $\text{op}_1 \in \{+, -, +=, -=, =, <, <=, ==, !=, >, >= \}$	$a \in \text{ROS-TYPED-VARS} \wedge$ $b \notin \text{ROS-TYPED-VARS};$ $\text{ROS-VAR}(a, u)$	$D(b, u) = 1 (0.95); D(b, u) \xrightarrow{0.95} P(b, u)$
	$\min(a, b)$ $\max(a, b)$	$a \notin \text{ROS-TYPED-VARS} \wedge$ $b \in \text{ROS-TYPED-VARS};$ $\text{ROS-VAR}(b, u)$	$D(a, u) = 1 (0.95); D(a, u) \xrightarrow{0.95} P(a, u)$
	$a = b \text{ op}_2 \cos(x), a = b \text{ op}_2 \sin(x), \text{op}_2 \in \{*, / \}$ Function definition: $f(\text{type } a, \dots)$, Function call: $f(b, \dots)$	$a \notin \text{ROS-TYPED-VARS} \wedge$ $b \notin \text{ROS-TYPED-VARS}$	$P(a, U) \xleftrightarrow{0.95} P(b, U)$
Naming	var_name	$p = 0.5 + \frac{1}{2} * \text{sim}(\text{var_name}, U)$	$N(\text{var_name}, U) = 1 (p); N(\text{var_name}, U) \xrightarrow{0.7} P(\text{var_name}, U)$
Computed-unit	$a = \text{expr}$	$\forall v: v \in \text{vars}(\text{expr}) \wedge$ $v \in \text{ROS-TYPED-VARS};$ $\text{cu} = \text{unit}(\text{expr})$	$C(a, \text{cu}) = 1 (1.0); C(a, \text{cu}) \xrightarrow{0.95} P(a, \text{cu})$
		$\exists v: v \in \text{vars}(\text{expr}) \wedge$ $v \notin \text{ROS-TYPED-VARS};$ $\text{cu} = \text{unit}(\text{expr})$	$C(a, \text{cu}) = 1 (0.8); C(a, \text{cu}) \xrightarrow{0.95} P(a, \text{cu})$
Known-symbol	$\cos(a), \sin(a)$		$K(a, \text{radian}) = 1 (0.95); K(a, \text{radian}) \xrightarrow{0.95} P(a, \text{radian})$
Conversion	$a = b * \pi / 180$		$F(a, \text{radian}) = 1 (0.9); F(a, \text{radian}) \xrightarrow{0.95} P(a, \text{radian}),$ $F(b, \text{degree_360}) = 1 (0.9); F(b, \text{degree_360}) \xrightarrow{0.95} P(b, \text{degree_360})$
	$a = b * 180 / \pi$		$F(a, \text{degree_360}) = 1 (0.9); F(a, \text{degree_360}) \xrightarrow{0.95} P(a, \text{degree_360}),$ $F(b, \text{radian}) = 1 (0.9); F(b, \text{radian}) \xrightarrow{0.95} P(b, \text{radian})$
	$a \text{ op num}, \text{op} \in \{+, -, +=, -=, =, <, <=, ==, !=, >, >= \}$	$\text{num: numerical value} > 2\pi;$ $\text{unit}(a) == \text{radian}$	$F(a, \text{degree_360}) = 1 (0.9); F(a, \text{degree_360}) \xrightarrow{0.95} P(a, \text{degree_360})$
	$a \text{ op } \pi, \text{op} \in \{+, -, +=, -=, =, <, <=, ==, !=, >, >= \}$		$F(a, \text{radian}) = 1 (0.95); F(a, \text{radian}) \xrightarrow{0.95} P(a, \text{radian})$

The confidence 0.7 for all naming constraints reflects that variable names are uncertain and can cause false unit inconsistencies if not augmented with other evidence. This confidence is the lowest among all confidence probabilities configured in *Phys*, as naming usually provides the weakest hint about a variable's unit. The value 0.7 was empirically identified, yielding a sufficiently high TP rate (> 80%) while retaining enough detection power to find significantly more unit inconsistencies than other methods.

Computed-Unit Constraints. Computed-unit constraints are collected for the assignment resulting from mathematical expressions, which may compose/compute new units from the operands' units. To generate a computed-unit constraint for a mathematical expression, we first need unit derivation rules. For example, for a division expression x/y , the unit derivation rule is:

$$\frac{P(x, u)}{\text{unit}(x) = u} \quad \frac{\text{unit}(x) = \text{meter}, \text{unit}(y) = \text{second}}{\text{unit}(x/y) = \text{meter-per-second}}$$

Where $\text{unit}(x)$ yields the unit of x based on the current unit assignment of x , and hence the units of x and y are 'meter' and 'second', respectively. Therefore the computed unit for x/y is 'meter-per-second'. Other derivation rules can be similarly defined.

Once we have computed the resulting units, we generate two constraints: 1) a prior probability constraint $C(\text{var}, \text{cu}) = 1(p)$, indicating that we observe var has a computed unit cu with probability p ; and, 2) $C(\text{var}, \text{cu}) \xrightarrow{0.95} P(\text{var}, \text{cu})$, that propagates the initial confidence to the unit assertion of variable var . An example of a

computed-unit constraint is shown in Table 2 (7). This constraint is generated once the units for the expression $\text{sqrt}(wx1 * wx1 + wy1 * wy1)$ is computed to be ms^{-1} , resulting in a computed-unit predicate $C(q1, \text{ms}^{-1})$ with confidence p .

The value of probability p depends on which variables contribute to a computed unit cu . If expr has only ROS variables, then a gets unit cu with probability 1.0, otherwise 0.8. After empirically exploring a range of values, the value of 0.8 is set higher than the naming hint confidence (0.7) and lower than the later discussed unit conversion hint confidence (0.9).

Known-Symbol Constraints. Software dealing with physical quantities often uses mathematical functions from some math library. For example, we observed a lot of usage of two such functions: $\sin(a)$ and $\cos(a)$. Both functions accept an argument that represents an angle expressed in 'radian'. We formulate this unit hint into two known-symbol constraints: 1) $K(a, \text{radian}) = 1(0.95)$; and, 2) an implication constraint $K(a, \text{radian}) \xrightarrow{0.95} P(a, \text{radian})$. The 0.95 probability models the uncertainty arising from a possible use of a variable with an incorrect unit assignment.

Conversion Constraints. Many robotic and cyber-physical programs reason about spatial relationships with angles, and developers use both 'radian' or 'degree_360'. Conversion constraints capture common expressions that convert between 'radian' and 'degree_360' and provide hints about units.

So, in the angle conversion expression $a = b * \pi / 180$, variable b should be of unit 'degree_360' and variable a should be

of unit ‘radian’. The generated conversion constraints would be: 1) $F(a, \text{radian}) = 1(0.9)$, $F(b, \text{degree_360}) = 1(0.9)$; and, 2) implication constraints: $F(a, \text{radian}) \xrightarrow{0.95} P(a, \text{radian})$, $F(b, \text{degree_360}) \xrightarrow{0.95} P(b, \text{degree_360})$. If a variable with unit ‘radian’ from the previous iteration is added or compared with 180, then we infer ‘degree_360’, as shown in Table 3. The observational confidence of 0.9 informs the prior probability constraints for these hints.

In addition, if a variable is added or compared with π , then we infer ‘radian’. This hint is formulated as a conversion constraint in Table 3, with the high probability of 0.95.

4.2 Probabilistic Inference Engine

Factor. Here, we discuss how the probabilistic constraints are translated into probabilistic functions. The functions serve as nodes in a factor graph. All the predicates present in constraints are represented as boolean variables. An implication constraint, $\text{pred1} \xrightarrow{p} \text{pred2}$, is translated into a factor $F(\text{pred1}, \text{pred2})$ as:

$$F(\text{pred1}, \text{pred2}) = \begin{cases} p, & \text{if } (\text{pred1} \rightarrow \text{pred2}) \text{ is true} \\ 1 - p, & \text{otherwise} \end{cases} \quad (5)$$

and, a bidirectional constraint, $\text{pred1} \xleftrightarrow{p} \text{pred2}$, is divided into two constraints: $\text{pred1} \xrightarrow{p} \text{pred2}$ and $\text{pred2} \xrightarrow{p} \text{pred1}$, which are then translated. A prior probability constraint $\text{pred1} = 1(q)$, is translated into a factor $F(\text{pred1})$ as:

$$F(\text{pred1}) = \begin{cases} q, & \text{if } (\text{pred1}) \text{ is true} \\ 1 - q, & \text{otherwise} \end{cases} \quad (6)$$

We denote a factor with a corresponding probabilistic constraint formulation, e.g. $F(\text{pred1}) : \text{pred1} = 1(q)$.

The boolean variables in Figure 3a together with the probabilistic constraints in Table 2 (1–7) can be translated into the factors shown in Figure 3b.

Factor-graph. A factor graph is a bipartite graph with two kinds of nodes: variable nodes and factor nodes. The edges join each factor with its variables, i.e., the variables over which a probabilistic function corresponding to the factor is defined. Figure 3c shows a factor-graph for the variables and factors from Figures 3a–3b. Factors F_{12} , F_{13} and F_{14} are omitted for simplicity.

Belief Propagation. We use the *sum-product* belief propagation algorithm [14] for probabilistic inference. It is an iterative algorithm that passes belief messages between adjacent nodes and updates the probability for each node based on the received messages. An updated probability is propagated to adjacent nodes in the next iteration. The algorithm terminates when the probabilities converge.

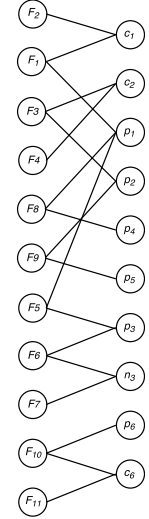
Iteration. *Phys* iterates during Stage 1, as shown in Figure 2. This iteration is critical to pick up additional constraints. For example, Table 2 on the right side lists the iteration number in which each probabilistic constraint was collected. As shown in the table, constraints (17) and (18) were only inferred after most of the other units in the program had been determined. In general, it is important to consider whether an iteration will reach a fixed point and terminate. Unlike a traditional dataflow analysis, units do not fit well into a lattice-based approach, and therefore we cannot use the *Ascending Chain Condition* [21] to guarantee a fixed point. Therefore we manually bound the iterations to 4, and observe that most programs we have analyzed reach a fixed point within this bound.

$p_1 : P(v_ref_x_ , ms^{-1})$	$p_6 : P(q1, ms^{-1})$
$p_2 : P(v_ref_y_ , ms^{-1})$	$c_1 : C(v_ref_x_ , ms^{-1})$
$p_3 : P(\text{linearSpeedX}Mps_ , ms^{-1})$	$c_2 : C(v_ref_y_ , ms^{-1})$
$p_4 : P(wx1, ms^{-1})$	$c_6 : C(q1, ms^{-1})$
$p_5 : P(wy1, ms^{-1})$	$n_3 : N(\text{linearSpeedX}Mps_ , ms^{-1})$

(a) Boolean variables representing the predicates

	Factors
(1)	$F_1 : c_1 \xrightarrow{0.95} p_1,$ $F_2 : c_1 = 1(1.0)$
(2)	$F_3 : c_2 \xrightarrow{0.95} p_2,$ $F_4 : c_2 = 1(1.0)$
(3)	$F_5 : p_1 \xrightarrow{0.95} p_3,$ $F_{12} : p_3 \xleftarrow{0.95} p_1$
(4)	$F_6 : n_3 \xrightarrow{0.95} p_3,$ $F_7 : n_3 = 1(0.7083334)$
(5)	$F_8 : p_4 \xrightarrow{0.95} p_1,$ $F_{13} : p_1 \xleftarrow{0.95} p_4$
(6)	$F_9 : p_5 \xrightarrow{0.95} p_2,$ $F_{14} : p_2 \xleftarrow{0.95} p_5$
(7)	$F_{10} : c_6 \xrightarrow{0.95} p_6,$ $F_{11} : c_6 = 1(0.8)$

(b) Factors



(c) Factor Graph

Figure 3: Factors and factor-graph for the probabilistic constraints (1)–(7) of our example in Table 2.

4.3 Complexity and Termination

Preprocessing builds a context-insensitive call graph, and topologically sorting this graph is $O(|V| + |E|)$, worst case $O(|E|^2)$ when removing cycles. Collecting probabilistic constraints involves at most h loops over each statement where h is the height of the statement’s AST. The probabilistic inference engine implements an approximate solution to the sum-product message passing algorithm [14] that is quadratic. Collecting probabilistic constraints and the sum-product are run within a loop bounded by a constant (four times). After the loop, detecting inconsistencies involves a linear scan of program variables and the program’s AST. Overall, complexity is quadratic in time and space. This approach terminates because we bound the loops to collect probabilistic constraints and run sum-product.

5 EVALUATION

Our main goal is to evaluate the effectiveness of *Phys* in both unit inference as well as unit-inconsistency detection. For that, we address the following research questions:

- **RQ₁.** How effective is our approach in physical unit type inference compared to the state of the art?
- **RQ₂.** Can our approach detect more unit-inconsistencies compared to the state of the art?
- **RQ₃.** How useful are various types of constraints defined in our approach?

We have implemented *Phys* in Python. It relies on a few third-party components: *Cppcheck* [17] is used to obtain an intermediate

form of a C++ program including a list of tokens, an AST for each statement and symbol tables for variables, functions, and scopes; *libDAI* [20] is used as the probabilistic inference engine. Moreover, as mentioned before, *Phys* utilizes a one-time ‘mapping’ of ROS data structures to physical units provided by *Phriky*. Our code is available for download at <https://zenodo.org/record/1310129>.

The experimental evaluation is conducted on sample C++ files picked from a large number of ROS-based projects (i.e., robotic software) publicly available on GitHub. We used 90 files for unit inconsistency detection in Section 5.2, and 30 files for type inference in Section 5.1 because of the manual annotation effort. The list of public software is at <http://www.ros.org/browse/list.php>. The execution time of *Phys* ranges from 1 to 28 seconds for a file with details elided.

For the comparison of *Phys* with the state of the art, we use *Phriky*. The robotics programs that we use for experiments do not come with any physical unit information. In order to determine whether a variable can have a unit and whether an inferred unit is correct, we manually collect the ground truth for all the reported variables in the sample test-suite files. For determining true positive (TP) cases of the reported inconsistencies, we manually examine each inconsistency by reviewing the corresponding source code.

5.1 Physical-Unit Type Inference

In the first experiment, we want to evaluate the ability of *Phys* to infer physical unit types for variables. Note that, we consider only those variables that can have physical units (some variables in robotics software do not represent any physical quantity, e.g., for loop index variable). Plus, variables of integer type are assumed to be dimensionless. Since *Phys* reports a ranked list of units for a variable, we consider only the top unit in this experiment.

Experiment Setup. We compute results for three categories of variables: 1) variables that *Phriky* could not assign any unit to, 2) variables that *Phriky* assigned an incorrect unit to, and 3) variables that *Phriky* assigned a correct unit to. Note that, *Phriky* sometimes assigns more than one unit to a variable due to the unit-resolution rule of performing union on an addition expression. In that case, if a unioned set contains a correct unit, then the unit assignment is considered as correct for *Phriky*. Also, variables of ROS data type, that obtain units from the ‘mapping’, are not included in the result computation. The experiment is performed on a sample test suite of 30 C++ files randomly picked from ROS-based projects.

RQ₁ Results: Unit Inference. Table 4 summarizes the results. Column ‘Total Vars (#)’ shows the total number of variables of non ROS data types for which *Phys* could infer units. Further, the table presents the count of variables in each category that *Phys* could infer units for in Columns ‘Var (#)’. Columns ‘Var (%)’ show the percentage of a total number of variables in a sample file that the corresponding ‘Var (#)’ accounts for. The accuracy of the unit assignments is shown in Columns ‘TP (%)’. The TP rate is computed as ‘TP%’=TP/‘Var(#)', where TP represents the count of true positive unit assignments (not shown in the table for brevity). Observe that, we achieve an overall TP rate of greater than 88% in each category. Also, it can be observed that *Phys* is able to infer units for a lot of variables that *Phriky* cannot. In particular, *Phys* infers units for 783 variables, whereas *Phriky* could assign units to only 302 variables.

Effect of Constraint Probabilities. As seen in Section 4.1, *Phys* is configured with the following parameters for constraints: 0.7 for naming, 0.8 for computed-unit and 0.9 for conversion. We performed a couple of additional experiments to study the effect of using different parameters values: 1) Naming probability: *Phys* was evaluated with four values, namely, 0.6, 0.7, 0.8 and 0.9. It was observed that it had negligible impact on unit-inference for the 30-files set; 2) Combination of computed-unit and conversion probabilities: as mentioned before, computed-unit probability is purposely chosen to be lower than conversion probability. Therefore, we evaluated *Phys* with three combinations of values for (computed-unit, conversion), namely, (0.8, 0.9), (0.8, 0.8) and (0.9, 0.8), i.e., lower than, equal to and higher than. It was observed that the TP rate was decreased for the last two combinations in one of the categories of variables (i.e., variables with incorrect units by *Phriky*). The decrease was due to incorrect unit inference for some of the angle variables, which were inferred to be ‘radian’ instead of ‘degree_360’.

5.2 Unit-inconsistency Detection

In this experiment, we evaluate the ability of *Phys* to detect unit-inconsistencies. Note that we consider only high-confidence inconsistencies. An inconsistency is considered high-confidence only if all the units in the inconsistent expression are known (no unknown units for variables, and no constants that may or may not bear an implicit unit). Both *Phriky* and *Phys* can be configured to report only high-confidence cases.

Experiment Setup. We compute the TP rate of the reported inconsistencies for both *Phys* and *Phriky*. Due to the substantial manual efforts entailed in identifying the ground truth for variables’ units, we selected only a subset (30 files) of a large number of ROS-based C++ projects as our sample test-suite for the previous experiment. However, it would be interesting to see how *Phys* performs on other files as well. Therefore, we divide the experiment into two parts. In part one, we compute the results for our sample test suite of 30 files used in the previous experiment. We call it the *30-files set*. In part two, we compute results for an expanded set of randomly selected sample files. For the selection of the expanded set, we ran *Phys* on 28,484 ROS-based projects’ files available on GitHub. *Phys* reported inconsistencies in 990 files (i.e. 3.5% of files with units). We then randomly selected 60 files for which inconsistencies were reported by *Phys* to form the expanded set.

RQ₂ Results: 30-Files Set Inconsistencies. Table 5 presents the TP and FP counts for each of the sample files in the 30-files set. The table does not show entries for files that are found to have zero inconsistencies by both *Phys* and *Phriky*. Columns 2-4 show the results for *Phriky*, whereas Columns 5-7 show the results for *Phys*. It can be observed that, though *Phys* has lower TP rate (96.43%) than that of *Phriky* (100%), it has a capability to uncover more inconsistencies. The highlighted rows indicate the cases of inconsistencies missed by *Phriky*, but detected by *Phys*. In particular, the result for the file `summit_xl_robot_control.cpp` demonstrates the detection of the addition unit-inconsistency described in the motivation section (Section 3). Also, observe that for the file `action.cpp`, the number of TP inconsistencies by *Phys* is less than that by *Phriky*. However, though less, we found that the root cause of all those captured inconsistencies is same, and thus, we do not actually miss the case of an incorrect usage of units in this file.

Table 4: Physical unit type inference by *Phys* compared with *Phriky*.

30-files set	Inferred units	Variables with no units by <i>Phriky</i>			Variables with incorrect units by <i>Phriky</i>			Variables with correct units by <i>Phriky</i>		
	Vars (#)	Vars (#)	Vars (%)	TP (%)	Vars (#)	Vars (%)	TP (%)	Vars (#)	Vars (%)	TP (%)
Perception.cpp	94	67	77.91	95.52	1	100.0	100.0	26	100.0	100.0
labbot_teleoperation_twist.cpp	3	1	50.0	100.0	0	-	-	2	100.0	100.0
QuadScripts.cpp	48	22	78.57	100.0	0	0.0	0.0	26	96.30	100.0
ard_node.cpp	20	4	66.67	100.0	0	-	-	16	100.0	100.0
traj_builder.cpp	105	48	81.36	87.50	6	100.0	100.0	51	100.0	100.0
motor_and_sensors_controller.cpp	10	7	100.0	57.14	0	-	-	3	100.0	100.0
simulation_functions.cpp	2	1	33.33	100.0	0	-	-	1	100.0	100.0
follow.cpp	9	4	50.0	75.0	2	100.0	100.0	3	100.0	100.0
motor_control_hc.cpp	16	5	83.33	80.0	0	0.0	0.0	11	100.0	100.0
placement_wrt_workspace_action_server.cpp	5	5	71.43	100.0	0	-	-	0	-	-
interpolater.cpp	11	2	15.38	100.0	6	100.0	100.0	3	100.0	100.0
collvoid_local_planner.cpp	55	42	75.0	88.10	2	50.0	0.0	11	100.0	100.0
vel_scheduler.cpp	31	14	93.33	71.43	1	100.0	100.0	16	100.0	100.0
simple_pose.cpp	21	13	81.25	100.0	2	100.0	100.0	6	100.0	100.0
base_driver.cpp	32	25	96.15	92.0	5	100.0	60.00	2	100.0	100.0
channel_controller.cpp	65	49	87.50	73.47	2	100.0	100.0	14	100.0	100.0
odometry.cpp	7	2	50.0	100.0	0	-	-	5	100.0	100.0
viconxbee.cpp	5	4	100.0	100.0	0	-	-	1	100.0	100.0
base_controller.cpp	22	19	86.36	73.68	0	-	-	3	75.0	100.0
trajectory_planner_ros.cpp	49	41	51.25	90.24	2	50.0	0.0	6	100.0	100.0
summit_xl_robot_control.cpp	104	68	93.15	100.0	23	85.19	100.0	13	92.86	100.0
summit_xl_waypoints.cpp	7	7	87.50	100.0	0	-	-	0	-	-
summit_xl_joint_state.cpp	0	0	0.0	0.0	0	-	-	0	-	-
summit_xl_joystick.cpp	3	3	100.0	0.0	0	-	-	0	-	-
action.cpp	30	24	68.57	87.50	0	-	-	6	100.0	100.0
twist_marker.cpp	2	2	50.0	100.0	0	-	-	0	-	-
twist_mux.cpp	2	2	40.0	100.0	0	-	-	0	-	-
twist_mux_diagnostics.cpp	4	4	100.0	100.0	0	-	-	0	-	-
navigating_jockey.cpp	8	5	71.43	100.0	1	100.0	100.0	2	100.0	100.0
turtlebot_example_node.cpp	13	10	90.91	100.0	0	-	-	3	100.0	100.0
Total	783	500	76.34	89.40	53	76.81	88.68	230	98.71	100.0

Table 5: Inconsistencies for the 30-files set.

Sample Test-Suite1	<i>Phriky</i> Inconsistencies			<i>Phys</i> Inconsistencies		
	Total (#)	TP (#)	FP (#)	Total (#)	TP (#)	FP (#)
labbot_teleoperation_twist.cpp	2	2	0	2	2	0
QuadScripts.cpp	4	4	0	4	4	0
ard_node.cpp	6	6	0	6	6	0
traj_builder.cpp	1	1	0	1	1	0
motor_and_sensors_controller.cpp	2	2	0	4	4	0
simulation_functions.cpp	1	1	0	1	1	0
follow.cpp	2	2	0	2	2	0
motor_control_hc.cpp	7	7	0	7	7	0
placement_wrt...action_server.cpp	1	1	0	3	1	2
collvoid_local_planner.cpp	2	2	0	2	2	0
base_driver.cpp	3	3	0	3	3	0
odometry.cpp	2	2	0	2	2	0
viconxbee.cpp	3	3	0	3	3	0
trajectory_planner_ros.cpp	3	3	0	3	3	0
summit_xl_robot_control.cpp	0	-	-	8	8	0
action.cpp	3	3	0	2	2	0
twist_marker.cpp	1	1	0	1	1	0
turtlebot_example_node.cpp	0	-	-	2	2	0
Total	43	43 [100.0%]	0	56	54 [96.43%]	2

There is one file, *placement_wrt_workspace_action_server.cpp*, for which *Phys* reported two FPs. Both are reported because of incorrect unit assignment of a variable *max_velocity*. The naming convention component of *Phys* identifies it as unit ‘meter-per-second’. However, in the program, this variable has been used as a maximum velocity value for both linear and angular velocities and thus can have either ‘meter-per-second’ or ‘per-second’ unit.

Table 6: Inconsistencies for the expanded set.

	<i>Phriky</i> Inconsistencies			<i>Phys</i> Inconsistencies		
	Total (#)	TP (#)	FP (#)	Total (#)	TP (#)	FP (#)
Unit-inconsistencies	78	75 [96.2%]	3	190	156 [82.1%]	34
Files	25	24	1	60	45	16

RQ2 Results: Inconsistencies for the Expanded File Set. Table 6 shows the summarized results for the expanded sample set. The overall TP rate for *Phys* is 82.1%. *Phys* detects 103.3% more inconsistencies compared to *Phriky*, including every inconsistency that *Phriky* detects. *Phys* finds 156 true positive inconsistencies in 45 files, whereas *Phriky* was able to find only 75 in 24 files.

Also, a number of FPs (34) are reported by *Phys*. They are generally caused by an incorrect unit inference of some variables, due to the inherent uncertainty modeled into the probabilistic inference approach. Majority of these FPs are caused by variables that are intentionally used to represent two different quantities (i.e. units) in a program, e.g., variable *run_vel* is used as both linear velocity and angular velocity. Other causes for FPs include: a) variable with a name reflecting a physical quantity, but used as a scalar (e.g. *width*); b) controller gain variables, which are used only in the controller equation and may carry implicit time quantity (i.e., it may have any of the ‘second’ or ‘per-second’ units or no unit).

5.3 Constraint Distribution

RQ3 Results: Constraint Usage. Here, we present a study on the types of constraints collected by *Phys* in our sample test suites. Table 7 presents a count of files for which a particular type of

Table 7: Usage of various constraint types.

	Files (#)				
	Dataflow	Naming	Computed-unit	Known-symbol	Conversion
30-Files Set	27	27	28	14	10
Expanded Set	57	56	52	35	20

constraint was collected. It can be observed that dataflow, naming, and computed-unit constraints play a major role in the unit-inference. Also, the other two types, known-symbol and conversion constraints, have been found in a number of files and thus are useful in strengthening the unit-inference.

6 THREATS AND LIMITATIONS

6.1 Threats

Self-labeling. One threat is that this effort uses self-labeled data to evaluate both physical unit type inference and inconsistency detection. To mitigate this threat when labeling physical unit types, we had multiple authors review the type assignments and also used *Phys* to show inconsistencies when a physical unit type needed correction. To mitigate this threat with inconsistencies, the authors evaluated inconsistencies independently and compared results.

Overfitting. By assuming English and encoding priors for suffixes like ‘speed’ that could mean either linear or angular velocity (different abstract types) there is a threat of overfitting. We mitigate this threat by using a small but general set of suffixes (41 entries) as described in Section 4.1 and observing that we evaluated *Phys* on 60 random files drawn from 28,484 files with inconsistencies, and observed only few FPs caused by incorrect suffix assumptions. **“Magic” Numbers.** We use three predefined confidence values for naming, computed-unit and conversion constraints, respectively. Their values are determined empirically and hence pose threats to our results. Our experiments show that the results are not that sensitive to the values.

6.2 Limitations

False Negatives. The number of false negatives in the dataset is unknown, so we cannot calculate recall. To address this limitation, we will examine evaluating the approach after seeding faults.

Generality. This approach relies on having some initial abstract type information for physical units, in our case the ROS shared message libraries. However, this approach could also leverage some gradual type information from developer annotations. While our evaluation focuses on ROS C++ software for impact, the technique is general for other robotic systems.

7 RELATED WORK

Abstract Type Inference. Guo *et al.* [6] proposed a dynamic, unification-based analysis for abstract type inference in Java programs to aid program comprehension. Likewise, we infer abstract types based on program interactions, but our work is static and we infer inconsistencies. *Ayudante* [8] uses dataflow to cluster variables into abstract data types, then leverages a WordNet [18] similarity metric to cluster by variable name; differences between the clusterings are reported as abstract type inconsistencies. Like *Phys*, *Ayudante* uses dataflow and variable names, but *Phys* uses probabilistic reasoning to account for the uncertainty present in using variable names in isolation. Also, we found poor results using WordNet

in the physical units type domain without context, since physical units types are highly dependent on a combination of local clues (like ‘speed’ meaning either linear and angular velocity). *Ayudante* more aligns with a traditional unification-based type systems.

Probabilistic Inference in Software. Dietz *et al.* used probabilistic inference to localize bugs [5] and we also seek to find bugs. However, our work collects evidence during static analysis while their work collects evidence from program traces. Probabilistic inference is used for type inference [27], specification extraction [4, 16], security [15], and reasoning about approximate computations [19]. We take an inspiration for our approach’s design, in particular, from the work on probabilistic type inference for Python [32]. The difference lies in that we detect physical unit inconsistencies and model hints specific to the problem, such as expression forms and ROS data types. Furthermore, units are not a predefined set and they can be composed by the code. Our technique is hence iterative.

Physical Units in Software. Many efforts have proposed support for physical units with language extensions [1, 11, 12], unit-annotation libraries [30], or dynamic techniques [29]. The tool *Osprey* [10] detects unit inconsistencies with static analysis by using developer annotations and propagating units through data-flow and constraints, but only works on Java programs. Our work is different from theirs in that we use information available in variable names and apply probabilistic constraints.

We target C++ code written for the Robot Operating System (ROS) [26], a popular open-source middleware. Robot software and ROS programs are used increasingly in both academic and industrial robots [28] and contain many variables measured in physical units. We build on *Phriky Units* ‘mapping’ [23, 24], a lookup table from shared library attributes to physical units, but assign more units to variables by adding additional constraints, allowing our approach to detect more inconsistencies. Further, our approach makes more variable assignments because it applies units after collecting constraints from the whole program, rather than *Phriky* that only makes a linear scan and cannot go backwards.

8 CONCLUSION

We have presented a novel probabilistic approach for abstract type inference of physical units and inconsistency detection in robotic systems. The approach leverages uncertain hints about variables’ units such as variables’ names, expression forms, and associated operations to make probabilistic inferences of unit types. We have implemented this approach as a tool *Phys*. *Phys* can infer units for 159% more variables than state-of-the-art, leading to the detection of more than 103% inconsistencies without additional developer effort, and with a true positive rate of 85%. In the future, we would like to address the causes for the reported false positives and incorporate more unit hints such as those present in equations that form a robot’s sensing, planning, and control components.

ACKNOWLEDGEMENTS

We thank our insightful reviewers. This research was supported by ONR contracts N000141410468 and N000141712947, NSF awards 1638099, 1526652, 1718040, 1748764, and 1409668. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

- [1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. 2005. The Fortress language specification. *Sun Microsystems* 139 (2005), 140.
- [2] BIPM. 2006. *Le Système international d'unités / The International System of Units ('The SI Brochure')* (eighth ed.). Bureau international des poids et mesures. http://www.bipm.org/en/si/si_brochure/
- [3] Percy Williams Bridgman. 1922. *Dimensional Analysis*. Yale University Press.
- [4] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for Data Structures. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 255–266. http://www.usenix.org/events/osdi08/tech/full_papers/cozzie/cozzie.pdf
- [5] Laura Dietz, Valentin Dallmeier, Andreas Zeller, and Tobias Scheffer. 2009. Localizing Bugs in Program Executions with Graphical Models. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada.*, Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta (Eds.). Curran Associates, Inc., 468–476. <http://papers.nips.cc/paper/3792-localizing-bugs-in-program-executions-with-graphical-models>
- [6] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. 2006. Dynamic Inference of Abstract Types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 255–265. <https://doi.org/10.1145/1146238.1146268>
- [7] S. Hangal and M. S. Lam. 2009. Automatic dimension inference and checking for object-oriented programs. In *2009 IEEE 31st International Conference on Software Engineering*. 155–165. <https://doi.org/10.1109/ICSE.2009.5070517>
- [8] Irfan Ul Haq, Juan Caballero, and Michael D. Ernst. 2015. Ayudante: identifying undesired variable interactions. In *Proceedings of the 13th International Workshop on Dynamic Analysis, WODA@SPLASH 2015, Pittsburgh, PA, USA, October 26, 2015*, Harry Xu and Walter Binder (Eds.). ACM, 8–13. <https://doi.org/10.1145/2823363.2823366>
- [9] Paul N. Hilfinger. 1988. An Ada Package for Dimensional Analysis. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 189–203. <https://doi.org/10.1145/42190.42346>
- [10] Lingxiao Jiang and Zhendong Su. 2006. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. 262–271. <https://doi.org/10.1145/1134323>
- [11] Michael Karr and David B. Loveman, III. 1978. Incorporation of Units into Programming Languages. *Commun. ACM* 21, 5 (May 1978), 385–391. <https://doi.org/10.1145/359488.359501>
- [12] Andrew Kennedy. 2009. Types for Units-of-Measure: Theory and Practice. In *Central European Functional Programming School - Third Summer School, CEFPS 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*. 268–305. https://doi.org/10.1007/978-3-642-17685-2_8
- [13] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press. <http://mitpress.mit.edu/catalog/item/default.asp?tttype=2&tid=11886>
- [14] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. Information Theory* 47, 2 (2001), 498–519. <https://doi.org/10.1109/18.910572>
- [15] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Discovering Semantic Data of Interest from Un-mappable Memory with Confidence. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society. <https://www.ndss-symposium.org/ndss2012/discovering-semantic-data-interest-un-mappable-memory-confidence>
- [16] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 75–86. <https://doi.org/10.1145/1542476.1542485>
- [17] Daniel Marjamaeki. 2013. Cppcheck - A tool for static C/C++ code analysis. <http://cppcheck.sourceforge.net/>
- [18] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (Nov. 1995), 39–41. <https://doi.org/10.1145/219717.219748>
- [19] Sasa Misailovic. 2017. Probabilistic reasoning for analysis of approximate computations. In *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2017, Seoul, Republic of Korea, October 15-20, 2017*. 4:1. <https://doi.org/10.1145/3125501.3125524>
- [20] Joris Mooij. 2010. libDAI - A free and open source C++ library for Discrete Approximate Inference in graphical models. <https://staff.fnwi.uva.nl/j.m.mooij/libDAI/>
- [21] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. <https://doi.org/10.1007/978-3-662-03811-6>
- [22] John-Paul Ore, Sebastian G. Elbaum, and Carrick Detweiler. 2017. 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, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*. IEEE, 712–718. <https://doi.org/10.1109/IROS.2017.8202229>
- [23] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Lightweight Detection of Physical Unit Inconsistencies Without Program Annotations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 341–351. <https://doi.org/10.1145/3092703.3092722>
- [24] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. 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 (ISSTA 2017)*. ACM, New York, NY, USA, 352–355. <https://doi.org/10.1145/3092703.3098219>
- [25] Judea Pearl. 1986. Fusion, Propagation, and Structuring in Belief Networks. *Artif. Intell.* 29, 3 (1986), 241–288. [https://doi.org/10.1016/0004-3702\(86\)90072-X](https://doi.org/10.1016/0004-3702(86)90072-X)
- [26] 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.
- [27] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. <https://doi.org/10.1145/2676726.2677009>
- [28] ROS Industrial Consortium. 2016. Current Members - ROS Industrial. <http://rosindustrial.org/ric/current-members>
- [29] G. Rosu and Feng Chen. 2003. Certifying measurement unit safety policy. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. 304–309. <https://doi.org/10.1109/ASE.2003.1240326>
- [30] Matthias Schabel and Steven Watanabe. 2010. Boost Units. http://www.boost.org/doc/libs/1_66_0/doc/html/boost_units.html
- [31] Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, et al. 2010. The F# 2.0 language specification. *Microsoft, August* (2010).
- [32] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 607–618. <https://doi.org/10.1145/2950290.2950343>