Lightweight Detection of Physical Unit Inconsistencies without Program Annotations

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

ABSTRACT

Systems interacting with the physical world operate on quantities measured with physical units. When unit operations in a program are inconsistent with the physical units' rules, those systems may suffer. Existing approaches to support unit consistency in programs can impose an unacceptable burden on developers. In this paper, we present a lightweight static analysis approach focused on physical unit inconsistency detection that requires no end-user program annotation, modification, or migration. 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. Then, leveraging rules from dimensional analysis, the approach propagates and infers units in programs that use these shared libraries, and detects inconsistent unit usage. We implement and evaluate the approach in a tool, analyzing 213 open-source systems containing +900, 000 LOC, finding inconsistencies in 11% of them, with an 87% true positive rate for a class of inconsistencies detected with high confidence. An initial survey of robot system developers finds that the unit inconsistencies detected by our tool are 'problematic', and we investigate how and when these inconsistencies occur.

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. Lightweight Detection of Physical Unit Inconsistencies without Program Annotations. In *Proceedings of 26th International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092722

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092722

1 INTRODUCTION

Systems that interact with the physical world operate on quantities measured in physical units. Consider a mobile robot that perpetually perceives the world through depth sensors, lasers, cameras, and gyroscopes, and interacts with the world through its actions. The robot collects measurements as it senses and acts, and transforms them into distances (`meters`) and angles (`radians`). The system also integrates these measurements with others such as time (`seconds`) to derive measures like the robot's velocity (`meters-per-second`).

To operate correctly, this kind of system must adhere to both the type semantics of the programming language and the unit semantics of the physical world. Consider the simple code snippet in Figure 1 belonging to the 'Romeo' robot [30]. The expression on line 191 calculates the distance between the current position and the goal. Normally this kind of distance function would add `meters-squared` to `meters-squared`, but this code incorrectly adds `meters` to `meters-squared`. The code compiles without complaint as both variables have the same programming type. Yet the inconsistency in how the units are combined in the code constitutes a fault that will go undetected by the type system, likely to manifest later as incorrect behavior.

The consequences of such unit inconsistencies in systems interacting with the physical world exhibit a range of severities, from mild to occasionally catastrophic [31]. There does not seem to exist, however, an authoritative estimate of how frequently unit inconsistencies occur or with what severity. Still, the related work in type systems indicates that these kinds of problems have been nagging system developers for a long time. As early as 1978, Karr and Loveman [16] advocated for the design of programming languages with support for unit types.

There are four kinds of approaches to detecting unit type inconsistencies: full native programming language support, migration to specialized type libraries, annotation-based approaches, and mining a program for contradictory variable type usage. Specialized language support for units is built into `Fortress` [1] and more recently `F#` [18]. The type library `boost::units` [29] offers static checking of unit type consistency but requires code migration for all variables involved in expressions with physical units. Annotation based methods [8, 13, 15, 32] commonly use dataflow and constraint solvers to reason about the units of unknown expressions. Approaches that mine variable type usage within a program to infer unit inconsistencies, such as UniFi [12], are annotation-free but require the program to contain unit usages that together are contradictory. The first three of these require system developers to incur an annotation or migration burden, and the last requires examples of contradictory usage to occur within a single program.

```
189 float computeDistance(geometry_msgs::Pose goal, geometry_msgs::Pose current)
190 {
191     float dist = (goal.position.x - current.position.x)*(goal.position.x - current.position.x)
        + (goal.position.y - current.position.y) + (goal.position.y - current.position.y)
        + (goal.position.z - current.position.z) + (goal.position.z - current.position.z);
```

meters squared

meters

Figure 1: Code snippet from SoftBank's Romeo robot [30] containing a unit inconsistency detected by our tool, subsequently acknowledged by the developers and patched. ***package*: ros-aldebaran *source*: https://git.io/v6Xll, *fixed source*: https://git.io/v6xkH**

In this work we seek to quickly detect unit inconsistencies without annotation or migration burdens, and without depending on the program to contain self-contradictory type usage. We present an approach that requires a one-time effort of building a mapping from attributes in shared libraries to units (instead of annotating every program that uses the shared library). Fortunately, middleware for cyber-physical systems often abstracts out commonly used quantities with physical units into shared libraries because these quantities are exchanged between components [19]. In addition to the mapping, the proposed approach uses *dimensional analysis*, [6] rules governing how physical quantities may be correctly combined, compared, and manipulated. As our approach analyzes a program, the mapping enables the automatic decoration of program variables with physical units, and applies rules from dimensional analysis to detect inconsistent usage of physical units. To keep our approach lightweight and practical, we accept design choices that compromise soundness and completeness. The work is contextualized and inspired in part by our observations and experiences with robotic systems where manipulating physical units is common, usually supported by libraries, and often challenging. However, this approach generalizes to systems that analyze, interpret, and reason about data with physical units, ranging from embedded systems to physical simulations of driverless cars, given that the system uses attributes in shared libraries that specify physical units. The key contributions of this work are:

- A novel approach for detecting unit inconsistencies in existing programs without extra effort from developers. It integrates a one-time mapping of shared libraries' attributes to physical units with a lightweight static approach implementing dimensional analysis.
- A lightweight static analysis tool implementing the approach for systems built on the Robot Operating System (ROS).[1] Available at http://nimbus.unl.edu/tools.
- An initial evaluation of our tool on a corpus of 213 systems containing +900, 000 LOC, wherein 24 systems (11%) contain unit inconsistencies, with a true positive rate of 87% for a class of unit inconsistencies detected with 'high confidence'.
- A validation of the kinds of unit inconsistencies detected by our tool, where 56% of inconsistencies were deemed problematic and 34% potentially problematic by surveyed developers of such systems.

[1]ROS is "maybe the most popular robotic middleware" [19], and has +3000 citations, +2500 systems, and nine million package downloads/month. We chose ROS deliberately for impact. An initial investigation of similar cyber-physical middleware indicates straightforward applicability to Orocos [7], OpenRTM [2], MOOS [4], and Yarp [21].

2 BACKGROUND AND MOTIVATING EXAMPLES

This section presents background on the nature of physical units and examples of unit inconsistencies. We first discuss how units are governed by the rules of dimensional analysis.

2.1 Background

Physical Unit Representation. Physical phenomena are quantified in terms of units, such as `meters-per-second` or `furlongs--per-fortnight`. More formally, units form an abelian group[2], and we extend the convention for units used by Jiang and Su [15] that models units as types and defines a simplified *unit type language*:

$$ut ::= meter \mid kilogram \mid second \mid ampere \mid kelvin \mid mole \mid radian \mid$$
$$degree \mid quaternion \mid candela \mid unity \mid ut_1 * ut_2 \mid ut^{-1} \mid \delta \quad (1)$$

The set of units we consider are the seven base units of the International System of Units (SI) [25]. The operator '$*$' means multiplication, *unity* is identity, ut^{-1} is a unit's inverse, and δ represents the unknown unit. We also include *radian*, *degree*, and *quaternion* because they are familiar to developers, even though they are equivalent to unity with dimensionless units `meter--per--meter` [22]. The seven base units can be combined to represent other physical quantities and these combinations are called derived units. For example, the Newton is the SI unit of force and is a derived unit. One Newton can be expressed in terms of its equivalent base units, $(\texttt{kilogram} * \texttt{meter}) * (\texttt{second} * \texttt{second})^{-1}$, or equivalently $kg\ m\ s^{-2}$. The unknown unit δ is useful in expressing and tracking uncertainty in units. The grammar ut generates the set of all possible unit assignments.

Consistent Unit Operations with Dimensional Analysis. Every base unit in the SI system corresponds to a base dimension. For example, the base unit `meter` has a base dimension of *length*. Other measurements of length, like `furlongs` or `smoots`, have different units than `meter` but the same dimension *length*. All quantities with units have a corresponding dimension. The rules governing how quantities with units can be manipulated is called *dimensional analysis* [6, 17]. Based on dimensional analysis, we define rules for addition, comparison, and assignment that are consistent only when satisfying the following:

Addition/Subtraction:

$$ut_1\{+,-\}ut_2 \rightarrow \{\text{consistent}\} \Leftrightarrow (ut_1 = ut_2) \quad (2)$$

Comparison:

$$ut_1\{<,>,\leq,\geq,=,\neq\}ut_2 \rightarrow \{\text{consistent}\} \Leftrightarrow (ut_1 = ut_2) \quad (3)$$

[2]Abelian groups are finite or infinite sets with a binary operation (for units, multiplication) that satisfy associativity, commutativity, closure, and have identity and inverse elements.

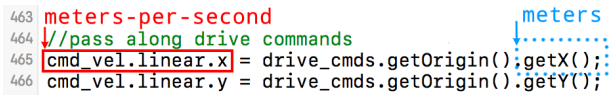

Figure 2: Inconsistent assignment.

*package:*ros-planning *source:*https://git.io/v6XlV

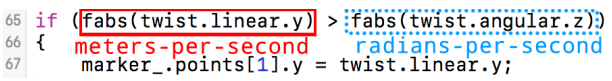

Figure 3: Inconsistent comparison.

*package:*ros-teleop *source:*https://git.io/v6Xld

For $ut_{1,2} \in ut$. Essentially, dimensional analysis specifies that you can only add or compare quantities with the same dimension. We extend this notion of consistency to programming languages: Assignment:

$$(ut_1 \leftarrow ut_2) \rightarrow \{\text{consistent}\} \Leftrightarrow (ut_1 = ut_2) \quad (4)$$

This specifies that assignment is only consistent when the units of a variable being assigned a new value are the same as the units being assigned. Operations that violate Eqs. 2-4 are called *unit inconsistencies* in this paper. We now explore three code examples to illustrate different kinds of units inconsistencies.

2.2 Motivating Examples

The code snippet shown in Figure 2 shows an assignment in line 465 with the variable `cmd_vel.linear.x` being assigned the value returned by the function `drive_cmds.getOrigin().getX()`. The Figure shows unit decorations to aid understanding. Both the variable and the value returned by the function have the data type `float64`, but they represent quantities with different units. The variable `cmd_vel.linear` is part of a structure called `Twist`, declared in a shared library `geometry_msgs`, that is specified to have units `meters-per-second`, while the function `getX()` on the right-hand-side (RHS) returns `meters`. Because the specified units are different than the units being assigned, this code does not satisfy Eq. 4 and is therefore unit inconsistent. We call this kind of unit inconsistency *assignment of multiple units*. As is, this code implicitly converts from one unit to another. At best, this inconsistency will make the code harder to maintain and understand. At worst, this implicit conversion might lead to unintended system behavior.

A second example is shown in Figure 3 line 65 where system developers compare two variables' magnitudes. The comparison is between `twist.linear.y` and `twist.angular.z`. The `Twist` data structure is defined in `geometry_msgs`, a shared library. The variable `linear.y` has units `meters-per-second` while the variable `angular.z` has units `radians-per-second`. This comparison does not satisfy Eq. 3 and is therefore inconsistent, and we call this *comparison of inconsistent units*. The system developer might have a reason to make this comparison, but such choices in code are suspicious and should be conspicuously documented and justified, especially for shared code.

Figure 4 shows a third example of unit inconsistency on line 1094 in an addition expression. This sums the squares of three quantities: `force.x`, `force.y`, and `torque.z`. The problem with this expression is that the units for `force` are different than the units for `torque`. Adding the square of `force` to the square of `torque` is not consistent by Eq. 2. We call this *addition of inconsistent units*. In the developer's defense, this calculation might behave as intended given input that implicitly normalizes these values. However, adding quantities with dissimilar units is generally devoid of physical meaning. Without explanation, this code might be considered a bewildering hack that works on one particular system, in one particular circumstance. If we assume this code is intentional, then the unit inconsistency reveals the existence of latent assumptions about the physical system. These assumptions hinder code re-use, since system developers must duplicate the system and environment or risk unintended behavior.

These examples illustrate how unit inconsistencies—multiple units, comparison/addition of different units—can result in programs that are difficult to understand and maintain, incorrect, or hard to reuse. These problems are discussed in more detail in our study in Section 5 that shows these inconsistencies lurk in at least 11% of the 213 systems we analyzed.

3 APPROACH

This section describes the approach, including how we perform a one-time mapping from class attributes in shared libraries to units, an algorithm that utilizes this mapping to detect unit inconsistencies, an analysis of the algorithm's complexity, and a discussion of the limitations of this approach. First, we describe requirements guiding the design decisions that shape our approach.

3.1 Requirements

The goal is to enable fast, lightweight, low-burden, meaningful unit inconsistency detection. More specifically, the requirements for an approach that meets this goal include:

- run on systems that manipulate standard physical units.
- execute sufficiently fast to be part of a build process.
- impose a minimal burden on system developers.
- detect inconsistencies that can lead to real problems.
- yield a low-enough false-positive rate to justify the value of its findings[3].

We pursued these requirements through a series of design and implementation iteration cycles in which we explored the tradeoffs between precision, recall, speed, and scalability. We now turn to the details of the approach resulting from these iterations.

3.2 One-time Mapping from Class Attributes in Shared Program Libraries to Units

The goal of mapping is to assign physical units to physical attributes in shared libraries. By *physical attributes* we mean attributes, structures, and class function return values found in shared libraries that represent quantities measured in physical units. Rather than annotating physical attributes at the point they are defined in shared libraries, this approach instead decouples this 'mapping' between physical attributes and units from the shared libraries. By decoupling the relationship between shared class attributes and physical units from the shared libraries, system developers do not need annotated copies of those libraries. Further, this avoids the reliance on unit-aware type libraries, compilers, or languages—all of which

[3]Both Bessey and Hovemeyer *et al.* used < 20% as a baseline [5, 14]

```
1094 abs_new_force = sqrt(   (kilogram-meter-per-second) squared
        (new_bubble_force.wrench.force.x * new_bubble_force.wrench.force.x) +
        (new_bubble_force.wrench.force.y * new_bubble_force.wrench.force.y) +
        (new_bubble_force.wrench.torque.z * new_bubble_force.wrench.torque.z));
                        (kilogram-meter-squared-per-second) squared
```

Figure 4: Inconsistent units during addition of force and torque in distance metric. *package*:eband_local_planner *source*: https://git.io/v6X8T

hinder re-use. When compared to individual system developers annotating program variables with physical units at declaration, this approach requires a single effort that can be broadly reused to enable unit inconsistency detection in every system that uses those shared libraries. This approach has larger benefits at larger scales. Overall, the purpose of mapping is to achieve the same effect as if the entire user base of the shared libraries were to agree to apply physical unit annotations in the shared libraries.

More formally, the mapping is a binary relation between two sets: the set of physical attributes PHYS_ATTRIB (where physical attributes are identified by fully qualified names (FQNs) in the shared libraries) and the set of unit types ut :

$$R_{mapping} \subseteq (\text{PHYS\_ATTRIB} \times ut) \quad (5)$$

Mapping Process. The mapping process involves four tasks:

- identifying shared libraries containing physical attributes.
- determining the physical unit for each physical attribute.
- finding the FQN for each physical attribute.
- encoding the mapping from FQN to physical unit in a structured form that can be used programmatically.

The first task, identifying the shared libraries, is a manual process that requires knowledge or analysis of the target domain. Identifying these libraries might be performed incrementally, to find the most commonly reused libraries within a corpus and then identifying physical attributes within that set. Or it might involve identifying a particular physical attribute and then examining the shared library that contains that attribute. The second task involves determining a physical attribute's units that might come from: 1) knowing the physical unit at design time; 2) finding source code comments or program documentation; 3) the class or variable name; and, 4) examining how this variable is used in context. The third task is simply recording the FQN for the shared class attribute. The fourth task, encoding the mapping, involves creating a lookup table from FQN to physical units, and choosing some structured form that can be read by an implementation. This lookup table is an implementation of the binary relation $R_{mapping}$.

Cost. The upfront effort to create the external mapping is slightly more than applying in-line annotations to physical attributes in shared libraries, because of the effort to encode the mapping in an external data structure that can be used programmatically. This additional effort is justified by the benefits mentioned above. Compared to annotating attributes in shared libraries, an external mapping introduces no reliance on unit-aware type libraries, compilers, or languages. When compared to annotating programs that use shared libraries, the single effort to create the mapping is much less than the repeated effort by every system developer to separately annotate program variable declarations for those shared libraries.

Mapping Example. To help illustrate the mapping process, we now provide an example instantiated within the cyber-physical domain, specifically, with a mapping built for the Robot Operating System (ROS). ROS is a widely-adopted middleware to enable rapid development in robotic systems used by both academic and professional developers, including industrial automation at Boeing [27] and autonomous driving at BMW [26].

The first step to identifying shared libraries with physical attributes in ROS was observing that ROS's component-based architecture defines data structures with physical units in shared libraries so components can exchange standardized data representing sensor measurements and actuator commands. We looked at these shared libraries and found physical attributes for navigation, geometric relationships, and sensor values. The shared library for navigation was `nav_msgs`, for geometric relationships the shared library was `geometry_msgs`, and for sensor values the shared library was `sensor_msgs`. We also determined the most frequently used libraries across the systems we studied, identifying additional libraries not in the standard code location of the ROS physical libraries. This was an iterative process, first finding one prospective shared library and then successively completing all the steps in the mapping process for that library before finding another shared library. Within these libraries, we found a variety of physical attributes such as `geometry_msgs::Twist.linear.y`, `sensor_msgs::Imu.angular_velocity_covariance`, and `nav_msgs::Odometry.linear.x`.

In the second step, we associate each physical attribute in the shared libraries with units. In the case of `Odometry::linear.x`, the documentation specifies this as 'velocity in free space'. Velocity has dimensions of `length-per-time` and the SI system is specified as the default units in ROS [10], therefore we assigned to it the units `meters-per-second`.

The third step of finding the FQN names of the physical attributes in shared libraries was straightforward, and involved copying the full name of the shared library, along with the names of the structures containing the physical attribute. An example is `nav_msgs::Odometry.linear.x`, where `nav_msgs` is the shared library, and `Odometry.linear.x` is the structure containing the physical attribute x.

The fourth step involved creating the encoding of the mapping, with a record or table entry for each pair of FQNs and corresponding units for each physical attribute in the shared library. An example of one entry is the FQN `nav_msgs::Odometry.linear.x` with the physical units $(\texttt{meters} * (\texttt{seconds})^{-1})$.

We repeated this process for other physical attributes, and collected a mapping of 246 total physical attributes (class attributes or function return values) from 82 classes across 7 shared libraries. These physical attributes mapped to 17 distinct derived units. Finally, we encoded the FQN of the physical attributes and its corresponding physical unit to create the mapping.

The effort to build the mapping for ROS was aided by the fact that two of the co-authors are proficient ROS users. Overall, we

completed the core mapping for ROS within 3-4 days. An initial investigation of similar cyber-physical middleware like Orocos [7], OpenRTM [2], MOOS [4], and Yarp [21] indicates that a mapping for these domains would require a similar effort.

Again note that this mapping is a one-time effort per shared library that then enables unit inconsistency detection in all systems that build on these shared libraries.

We now present an algorithm for unit inconsistency detection utilizing this mapping.

3.3 Algorithm for Lightweight Detection of Unit Inconsistencies

This section describes the algorithm LightweightDetectUnitInconsistency (Algorithm 1), that uses the mapping described in Sec. 3.2. Some functions of Algorithm 1 that require further explanation are described in the text below.

As fitting our requirements, this approach seeks the simplest analysis still capable of detecting meaningful unit inconsistencies. Our analysis is semi-flow-sensitive (a simplified forward dataflow), path-insensitive, context-insensitive, and intra-procedural. Note that although the analysis is intra-procedural, for some function calls the approach can determine the units of the return value because of the order the functions are analyzed. In these cases, the approach applies the units returned by the function at its call point.

A dataflow analysis is often defined using states, a transfer function, a lattice, and a join operation. The states represent knowledge at entry/exit points of blocks, a transfer function calculates changes to the state during that block, the lattice represents all possible abstract states arranged in a power-set hierarchy, and the join function calculates the state at the entry to a block by 'joining' the states that flow into that block in the control flow graph. In contrast, our analysis has only one single state, *State*, that enters and exits every statement. *State* is a set of tuples representing variable unit assignments, $\{(var, \{units\}), \ldots\}$ where $var \in$ VAR, the set of program variables and $\{units\} \subset ut$, the unit type language of Equation 1. A power-set lattice representation of the abstract state is a poor fit because physical units form an abelian group, and therefore we instead use a unit type language (Equation 1). Statements are analyzed sequentially without regard to control flow. At a program point, the units of a variable in *State* are the union of: 1) any units specified by the mapping because the variable is of a type that belongs to a shared library and represents a physical class attribute; 2) previous unit assignments. The transfer function from before a statement (the '*in*' state) to after the statement (the '*out*' state) is the union of: 1) the previous state; 2) the evaluation of the units resulting from the RHS expression of assignment and return statements. Since there is only one state, the join operation is unnecessary.

Overview. Algorithm 1 takes as input a program P and relation $\mathsf{R}_{\text{mapping}}$ from Sec. 3.2. During the loop in lines 5-10, the algorithm processes each program statement once. It detects unit inconsistencies in two ways: 1) within a statement for addition/comparison inconsistencies; and 2) by analyzing variables in the final version of *State* for multiple unit assignments to one variable.

Preprocess. In line 4, the algorithm preprocesses program P by constructing a context-insensitive call graph (without alias analysis) and performing a reverse topological sort, to analyze functions bottom-up. If the call graph contains a cycle, an edge of the cycle is removed from the call graph until no cycles are found. If the topological sort yields a partial order, the approach breaks ties arbitrarily and examines only the first ordering for simplicity. The output is an ordered list of functions.

Algorithm 1 Lightweight physical unit inconsistency detection over program P

Input: Program P and unit mapping $\mathsf{R}_{\text{mapping}}$.
Output: Set of unit inconsistencies.

```
1:  function LightweightDetectUnitInconsistency(P, R_mapping)
2:      UI ← ∅                                          ▷ Unit Inconsistencies
3:      State ← ∅
4:      sortedFunctions ← Preprocess(P)
5:      for function ∈ sortedFunctions do
6:          for statement ∈ function do
7:              DecorateWithUnits(statement, State, R_mapping)
8:              EvaluateExpressions(statement)
9:              UI ← UI ∪ DetectExpInconsistency(statement)
10:             State ← State ∪ TransferFunction(statement)
11:     UI ← UI ∪ DetectMultipleUnitInconsistencies(State)
12:     return UI

13: function TransferFunction(statement)
14:     newUnits ← getRHSUnits(statement)
15:     if newUnits = ∅ then
16:         return ∅
17:     if isAssignment(statement) then
18:         return {(getLHSVar(statement), newUnits)}
19:     else if isReturn(statement) then
20:         return {(functionName, newUnits)}
21:     return ∅
```

DecorateWithUnits. In line 7, this function traverses a statement's Abstract Syntax Tree (AST) and applies unit decorations to variables, when possible. We assume the existence of a relation between the set of program variables VAR and the set of physical attributes PHYS_ATTRIB:

$$\mathsf{R}_{\text{typeOf}} \subseteq (\text{VAR} \times \text{PHYS\_ATTRIB}) \tag{6}$$

This relation is commonly provided by a compiler front end, and in our tool this is provided by CPPCheck [20]. Using the composition of this relation with the mapping from Eq. 5 we have:

$$\mathsf{R}_{\text{unitsOf}} \equiv (\mathsf{R}_{\text{mapping}} \circ \mathsf{R}_{\text{typeOf}}) \subseteq (\text{VAR} \times ut) \tag{7}$$

Where $\mathsf{R}_{\text{unitsOf}}$ is the composition of the relations in Eq. 5 and Eq. 6 linking program variables to units.

Program variables can be decorated with units from either a prior assignment statement listed in *State* or when the variable's type is found in $\mathsf{R}_{\text{unitsOf}}$. The function DecorateWithUnits first checks for units in *State* and if no units are found, checks $\mathsf{R}_{\text{unitsOf}}$. If neither structure yields units, then the variable is decorated with δ, the unknown unit. An example of unit decoration using $\mathsf{R}_{\text{unitsOf}}$ is shown in the dotted boxed of Figure 5. These variables can be decorated because their variable type belongs to the shared library `geometry_msgs` that declares a class `WrenchStamped` with physical class attributes included in $\mathsf{R}_{\text{mapping}}$. The composed relation $\mathsf{R}_{\text{unitsOf}}$ connects variable `force.x` to the units `kg m` s^{-2}.

EvaluateExpressions. This function visits a statement's AST and attempts to resolve the units of expressions using the unit resolution rules shown in Table 1. It works from the leaves up, matching expressions to unit resolution rules and decorating the interior nodes of the AST with units. It continues to apply unit resolution rules in a loop until no changes are made. These rules apply when variables or expressions with units are combined and manipulated.

Note an important difference between the rule for multiplication and the one for addition: during multiplication, if one operand has known units but the other is δ, the unknown unit, we pessimistically assume the result is unknown; during addition, if one operand is known and the other is δ, we optimistically assume the result is the known unit. The reason multiplication is pessimistic is that there is only one way for multiplication to yield the same units, and many ways for the result to be different. Multiplication only yields the same units when multiplied by a scalar with *unity* as the unit, and assuming that every unknown variable involved in multiplication is a scalar leads to many false positives. The reason addition is optimistic is that the resulting sum must have the same units as the known operand or be inconsistent, and we cannot conclude the sum is inconsistent because δ is unknown.

An example of how the function EvaluateExpressions works is shown in Figure 5. The units in the dotted boxes were applied in DecorateWithUnits, and the three multiplications near the bottom of the AST match the multiplication rule in Table 1. By the multiplication rule, we add the exponents of the units of the operands, yielding the unit decorations on the three '$*$' symbols. Next, the rules match the '+' symbol up the tree, and apply the addition resolution rule, yielding the union of the operands' units. This function continues to apply unit resolution rules until no more changes can be made. This function only adds additional unit decorations and does not detect unit inconsistency in the expressions, which happens in the next function.

DetectExpInconsistency. This function applies the unit consistency tests from Eq. 2 (addition) and Eq. 3 (comparison) to expressions within a single statement. This function scans a statement's AST looking for inconsistencies like those in Figures 1,3,4, and 5. The example in Figure 5 shows an unit inconsistency detected while evaluating an addition expression.

As shown in Table 1, the unit inconsistency detection has a 'confidence' that can be either high or low if the expression contains δ, the unknown unit. Figure 5 shows the detection of inconsistent addition of $\mathtt{kg}^2\mathtt{m}^2\mathtt{s}^{-4}$ to $\mathtt{kg}^2\mathtt{m}^4\mathtt{s}^{-4}$ with high confidence.

TransferFunction. The transfer function in this analysis can only add new information to the state, and only for assignment or return statements. For assignment statements, the function getRHSUnits at line 14 simply returns the units decorating the '=', and otherwise returns the empty set. In line 10, *State* is updated as the union of *State* and the output of TransferFunction.

DetectMultipleUnitInconsistencies. Scanning *State* at line 11 of Algorithm 1 can reveal *assignment of multiple units* inconsistencies. This kind of inconsistency comes from two sources 1) variables assigned units contrary to their specification in the $\mathrm{R}_{\mathrm{mapping}}$; and 2) variables assigned different units at different points in the program.

When *State* contains multiple units for a variable this function reports inconsistencies with either low or high confidence, based on the presence of δ in a variable's units. This function reports high confidence if at least two units without δ are assigned to a program variable.

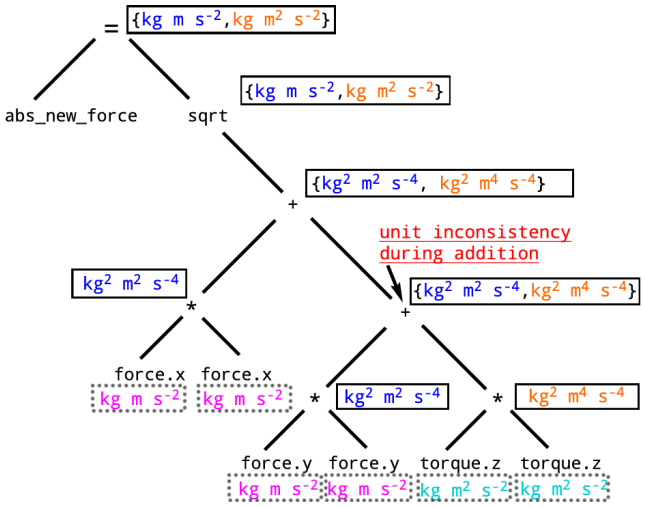

Figure 5: Example of a statement's AST from the code in Figure 4 with the shared class fully qualified name `WrenchStamped::wrench` omitted for simplicity. Figure shows unit decoration of variables by the relation $\mathrm{R}_{\mathbf{unitsOf}}$ (dotted boxes), and evaluation of expressions' units toward the root by unit resolution rules in Table 1 (solid boxes).

3.4 Termination and Complexity

Preprocessing requires a linear pass over the program to construct the context-insensitive call graph, and topologically sorting the call graph is $O(|V| + |E|)$ with a worst case $O(|E^2|)$ when detecting and removing cycles. The loop in lines 5-10 analyzes each statement once and is linear in the size of the input program AST. Decorating a statement's variables with units, evaluating expressions, detecting expression unit inconsistencies, and the transfer function are linear in the size of a statement's AST. After the loop, detecting multiple unit inconsistencies requires a linear scan of *State*, and *State* is as large as the number of program variables. Putting it all together, the worst case for the algorithm is quadratic in time and space.

Termination is guaranteed because the while loop within EvaluateExpressions applies unit resolution rules at most h times where h is the height of a statement's AST.

3.5 Limitations

While designed to be as fast and lightweight as possible while detecting useful inconsistencies, this design has limitations in applicability, soundness, and completeness. The mapping process can only apply units to physical attributes that are identified and correctly assigned units beforehand, and the number of program variables that can be labeled with unit information might be limited, bounding applicability. This approach is unsound because it includes infeasible sets of variable-unit assignments in *State* as it ignores control flow. This approach is incomplete because *State* misses some variable-unit assignments in loops and because it is not path-sensitive. Also, as described, the approach is semi-flow-sensitive and intra-procedural, only supports the mapped libraries,

Table 1: Unit resolution rules used in Algorithm 1 function EvaluateExpressions on line 8, and the inconsistency rules are used to detect addition/comparison inconsistencies in function DetectExpInconsistency on line 9.

EXPRESSION	CONDITION	RESULT	INCONSISTENT	CONFIDENCE
$ut_1\{*, \div\}ut_2$		ut_3 =add/subtract exponents of ut_1 and ut_2		
$ut_1\{*, \div\}\delta$		$ut_1 * \delta$ (pessimistic)		
$ut_1\{+, -\}ut_2$	$ut_1 = ut_2$	ut_1		
$ut_1\{+, -\}ut_2$	$ut_1 \neq ut_2$	$ut_1 \cup ut_2$	Yes, by Eq. 2	high
$ut_1\{+, -\}\delta * ut_2$	$ut_1 \neq ut_2$	$ut_1 \cup \delta * ut_2$	Yes, by Eq. 2	low
$ut_1\{+, -\}\delta$		ut_1 (optimistic)		
$pow(ut_1, n)$	$n \in \mathbb{R}$	multiple each ut_1 exponent by n		
$sqrt(ut_1)$		divide each ut_1 exponent by 2		
$sqrt(\delta)$		δ		
$ut_1\{<, >, \geq, \leq, \neq\}ut_2$	$ut_1 = ut_2$	none		
$ut_1\{<, >, \geq, \leq, \neq\}ut_2$	$ut_1 \neq ut_2$	none	Yes, by Eq. 3	high
$ut_1\{<, >, \geq, \leq, \neq\}\delta * ut_2$	$ut_1 \neq ut_2$	none	Yes, by Eq. 3	low
$ut_1\{<, >, \geq, \leq, \neq\}\delta$		none		
$\{floor, ceil, (f)abs\}(ut_1)$		ut_1		
$\{min, max\}(ut_1, ut_2)$		$ut_1 \cup ut_2$		
$\{min, max\}(ut_1, \delta)$		ut_1		
$(Boolean)\ ?\ ut_1\ :\ ut_2$		$ut_1 \cup ut_2$ (ternary operator)		

and does not handle complex dynamic concerns like dynamic dispatch, pointers, or Standard Template Library data structures. Further, the approach does not attempt symbolic analysis that could reason about statements like $(UNIT^n)^{(-n)}$. We expect to keep examining these design decisions, interleaving the incorporation of stronger analyses with an assessment of their impact in the overall cost-effectiveness and performance of the approach.

4 PROTOTYPE IMPLEMENTATION

Our implementation detects unit inconsistencies in C++ code written for ROS. The architecture follows the approach, is implemented in 3300 lines of python, and can be run from the command-line.

Our tool utilizes CPPCheck as a C++ preprocessor and parser [20], invoked with default parameters and includes directories: `cppcheck --dump -I ../include myfile.cpp`. The *dump* option generates an XML file containing: 1) every program statement as a separate abstract syntax tree; 2) a token list; and 3) a symbol database including functions, variables, classes, and all scopes. CPPCheck can explore multiple compilation configurations (different `#define` values), but in the reported results we only consider the default system configuration.

We use a visitor pattern in each statement's AST to apply units and evaluate expressions with unit resolution rules. During implementation, we realized that `radian` and `quaternion` require special handling: during multiplication, `radian` and `quaternion` act as *unity* since their units are `meters-per-meter`; during addition, they are 'coherent units of measure' [22], meaning that they cannot be added to a dissimilar unit, even though they are dimensionless.

An example inconsistency message for the code in Figure 4 reads: `Addition of inconsistent units on line 1094 with high confidence. Attempting to add` $kg^2m^2s^{-4}$ `to` $kg^2m^4s^{-4}$. We consolidate error messages to report only the first unit inconsistency for a particular variable.

Table 2: ROS Open-Source Repositories

CORPUS SOURCE	# of REPOSITORIES
Total ROS.org "Indigo" Projects Links	2416
Live Git Repos	649 of 2416
Git REPOS with C++ FILES	436 of 649
REPOS with C++ FILES AND ROS UNITS	213 of 436

We provide our prototype as a software artifact that can be downloaded from http://nimbus.unl.edu/tools.

5 EVALUATION

The goal of the evaluation is to assess the ability of the tool to detect unit inconsistencies that may cause problems for real systems. To achieve that goal we employ a methodology that captures two perspectives: 1) we examine the results of running our tool on a corpus of publicly available robotic systems and then hand-label the results as True and False Positives; and 2) we conduct a small survey of robotic system researchers to gauge whether they deem the unit inconsistencies detected by our tool to be 'problematic.' Finally we discuss threats to validity.

5.1 Analysis of Robotic Software Corpus

To evaluate the effectiveness of our tool we exercised it on a wide range of publicly available robotic software, specifically systems designed to work with the robotic middleware ROS.

The maintainers of ROS publish a list of public software repositories using ROS in academic and industrial robots. The list, published at http://www.ros.org/browse/list.php, includes projects at various stages of development, and for a wide variety of purposes: mobile robot navigation, collision detection libraries for robotic arms, drivers for depth cameras, control software for flying robots—a diverse set.

Table 2 shows statistics about this corpus. At the time we gathered this corpus, there were 2416 projects linked from the 'Indigo' version of ROS. Of these 2416 links, 649 were linked to live Git repositories. ROS supports C++, Python, and a few projects with LISP

and Java, but the majority are in C++, so we focused on those. Of the 649 live Git repositories, 436 contained C++ files. Of these 436, we found 213 repositories with systems containing shared libraries with physical attributes. For this initial work, we proceed with all ROS `geometry`, `navigation`, `transform`, `sensor`, and `time` libraries. The list of the 213 systems with GitHub download links and version numbers is available at http://nimbus.unl.edu/tools.

Scale and Speed. We ran our tool on 213 systems containing ROS physical units, analyzing 934,124 non-blank non-commented lines of C/C++ as reported by CLOC [9]. Analyzing all systems took 108 minutes (61 minutes to parse the files with CPPCheck and 47 minutes to perform our analysis), with an average analysis time of 31 seconds per system, when running on a MacBook Pro ('early 2015') with a 2.9 GHz Intel Quad Core i5 processor, and 16GB of memory. We only utilized a single core during evaluation, although this could be easily parallelized since the files and analysis are independent.

Effectiveness. We individually examined each inconsistency reported by the tool, reviewing the source code surrounding each reported line, and labeled each one as either 'True Positive' (TP) or 'False Positive' (FP). Note that labeling inconsistencies as TP or FP lets us calculate precision, but the number of 'False Negatives' (FN) is unknown and therefore we cannot calculate recall. This labeling process required several rounds of iterations as the analysis of some inconsistencies led us to question and re-analyze previous labels. The process converged towards the recognition of TP caused by addition/comparison of inconsistent units and assignment of multiple unit inconsistencies. We observe that multiple unit inconsistencies have distinct causes: 1) variable re-use (like temp variables); 2) disagreement between the units defined in the mapping (from the documentation) and the actual units used; 3) the previous two combined—variables with units from the mapping used as temporary variables. We currently do not distinguish between these causes but believe they could be separated automatically by observing whether the units come directly from the mapping and whether they are assigned only one kind of unit in the program.

Our results are summarized in Table 3. The overall TP rate, computed as $TP\% = 100 * TP/(TP + FP)$, for 'high confidence' unit inconsistencies is 87.0%. This includes the three types, variables assigned multiple units, addition of inconsistent units, and comparison of inconsistent units. As we noted earlier, for one of the cases where we contacted the authors of the code for clarification, shown in Figure 1, the inconsistency was acknowledged as a fault and developers patched the code immediately after our inquiry. Within the 'high confidence' TP, we found a TP rate of 84.6% for variables assigned multiple units. The False Positives with 'high confidence' all detect redundant implementations of vector cross-products and outer-products that are already provided by the ROS API, where `meters-squared` *intentionally* equals `meters`. Figure 6 contains one such case in line 90, which is frequently used and deemed correct by system developers. In general, our tool handles vectors like any other quantity and detects inconsistent addition, comparison, and assignment. We believe we could modify the tool to detect and ignore this special case, but we would have to be careful not to blind our tool to *unintentional* assignments of `meters-squared` to `meters`, therefore for now we accept these kinds of FP.

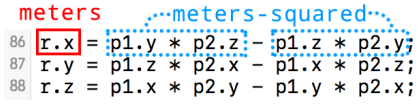

Figure 6: False positive during cross-product.
package:pr2-navigation *source*:https://git.io/v6xS7

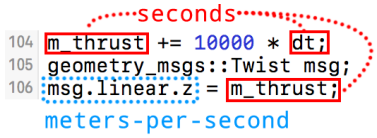

Figure 7: False positive with constant.
package:crazyflie_ros *source*:https://git.io/v6x7T

The overall TP rate for 'low confidence' unit inconsistencies is 37.45%, with about 50% more low confidence TP (64) than high confidence TP (40). The low TP rate is caused mainly by the large number of variables and constants with implicit units not found in the mapping. Figure 7 shows an example of a constant with implicit units. The constant 10000 in line 104 is part of a expression with units $\delta *$ `seconds` that is then assigned to *m_thrust*. The units assigned at line 104 flow to the usage of *m_thrust* in an assignment statement on line 106. Since the units for `msg.linear.z` are `meters-per-second` and it is assigned units of $\delta *$ `seconds`, our tool reports a low confidence unit inconsistency. In this case, the source of the FP is the constant 10000. Constants are basically devoid of units and we plan to assess them in more detail in the future.

5.2 Survey

We conducted a survey to obtain an initial assessment of whether these kinds of unit inconsistencies are problematic to robotic software developers. Specifically, some unit inconsistencies, like variable reuse and using a physical attribute to store a quantity against its specification, might be poor programming style, but also might not warrant a high-priority bug report. Therefore we wanted to assess the severity of these kinds of inconsistencies. Our survey instrument consists of eight questions, each showing a code example similar to those in Figures 2-4, drawn from unit inconsistencies detected by our tool. For each code example, we asked *"Is the unit inconsistency on line [X] problematic (e.g., cause failures, increase cost of maintenance, make code more difficult to understand, or introduce interoperability problems)?"*, with a choice of responses: 'yes', 'maybe', and 'no'. After each question, the respondents could add an open-ended explanation. The order of the questions was randomized for each respondent.

The target population for our survey included either heads of academic robotics research labs or their senior research associates. These labs publish regularly in top robotic conferences and use ROS extensively. We sent our survey to ten labs and received ten responses from six of the labs. We recognize the sample population size is small and may not generalize, but at this stage of the work we wanted quick, initial feedback before attempting a larger, more nuanced study that might be justified in the future.

Table 3: Classification of unit inconsistencies found by our tool. Note: this table presents precision and not recall, because recall requires false negatives (FN) that are unknown in our corpus.

	High Confidence			Low Confidence		
INCONSISTENCY TYPE	TP	FP	TP%	TP	FP	TP%
TOTAL	40	6	87.0%	64	107	37.4%
ASSIGNMENT OF MULTIPLE UNITS	33	6	84.6%	55	83	39.9%
ADDITION OF INCONSISTENT UNITS	5	0	100%	9	20	31.0%
COMPARISON INCONSISTENT UNITS	2	0	100%	0	4	0%

Table 4: Summary of survey responses to whether unit inconsistencies found by our tool are 'problematic.'

Question #	YES	MAYBE	NO
1	6	2	2
2	8	1	1
3	3	5	2
4	9	0	1
5	6	3	1
6 (Figure 2)	2	8	0
7 (Figure 4)	7	3	0
8	4	5	1
TOTALS	45	27	8
%	56%	34%	10%

Our results are shown in detail in Table 4. Overall, 56% of responses indicate that 'yes', these unit inconsistencies are problematic. The 'yes' responses included explanations from *'The addition of different units means nothing in real world'* to *'just bad programming.'* This fits with our assessment that many unit inconsistencies require attention or at least special explicit justification.

The 'maybe' responses (34%) included explanations, such as *'If the angular radius is unity, then OK, otherwise could lead to error'*, identifying a special case when the code *could be correct*, or *'I don't know when you'd like to compute this.'* Several 'maybe' responses indicated the possibility of a special circumstance when the unit inconsistency might not be problematic. In these cases our tool indicates a possible constraint on the circumstances under which the code behaves correctly, and for the unit inconsistencies detected by our tool, these special circumstances were never mentioned in the code comments, to our knowledge.

Of 'no' responses (not problematic), half came from one respondent (4 of 8), who explained: *'The problem I see is that the proposed method will get hung up in hacks that actually are workable solutions and it might be impossible for the average coder to fix these issues.'* We contend that detecting 'workable' 'hacks' is still valuable, especially for junior developers lacking the hard-earned experience necessary to recognize them in the first place.

Questions 6 and 7 from Table 4 of the survey are also presented in this work as Figure 2 and 4. Notice for question 6 that most respondents said 'maybe', and this code example shows unit inconsistency by assignment to a data type with a different physical unit specification, which is perhaps more an issue of code maintenance and reuse since it only uses a technically incorrect data container. However in question 7 (Figure 4) most respondents said 'yes problematic', and this code example contains addition of inconsistent units, which is perhaps more concerning because it might be incorrect. We believe that identifying both of these kinds of unit inconsistencies has value to system developers.

At the end of the survey we let respondents write an open-ended 'overall' feedback to these kinds of unit inconsistencies. The most critical respondent stated *'Overall a lot of unit inconsistencies will happen for control or optimization reasons and sometimes ... cannot be avoided,'* while the most laudatory stated *'This tool is amazing! At the very worst, it find out questionable programming practice that needs additional documentation. Most of the time, it finds bugs or hacky heuristics.'*

Overall, is spite of the limited size our population, and that this population does not represent industrial system developers, most responses affirm our assertion that the kinds of unit inconsistencies detected by this approach are problematic.

5.3 Threats to Validity

Self-labeling. One significant threat to the validity of our findings is our reliance on self-labeled TP and FP. For high confidence TP, it was relatively easy to see when and how the physical units are inconsistent, and for high confidence FP there were only a few mathematical corner cases like the cross and outer vector product that required careful consideration but we ended up confident about those classifications. Low confidence FPs were more straightforward to identify and often were attributable to a single variable with unknown units, such as the code example in Figure 7. Low confidence TPs were more time consuming to identify because these unit inconsistencies involved partial information, and we assumed the low-confidence inconsistencies were FP until proven to be a TP.

We mitigated the previous threat to internal validity by reviewing the results multiple times and discussing examples with other system developers, and ultimately by combining the analysis of the code corpus with the preliminary survey. The respondents classified most examples as either 'yes' or 'maybe' problematic, and provided many compelling justifications for their responses. To a large degree, these responses matched our expectations, but clearly further study is needed to better understand the larger body of inconsistencies found.

False Negatives. We cannot measure recall because the number of false negatives in the software corpus is unknown, but we intend to address this threat by seeding faults to better evaluate our approach. To our knowledge, there exists no dataset of labeled physical unit inconsistencies, although this work identifies an initial set of TP.

Cost-effectiveness. We recognize that this approach's cost effectiveness will vary across systems depending at least partly on the extent of physical unit usage, their degree of manipulation, and their standardization. Still, we argue that this threat to external validity is mitigated by the fact that most physical systems regularly

incorporate sensors and actuators that utilize standardized middleware libraries shared across platforms, all using some convention for data structures containing physical units.

Generality. Similarly, we recognize that we built the mapping for a particular robotic domain, namely ROS, but other mappings to a variety of middleware systems would help validate the generality of the approach. We are currently exploring other middleware, and will likely next target Orocos [7], OpenRTM [2], MOOS [4], and Yarp [21]. Still, ROS is by far the most commonly adopted robotic middleware, and has a growing user base.

Comparison to Other Tools. A further threat to external validity is that we do not directly compare our tool to other unit inconsistency detection tools. We do discuss these and other approaches in related work. A practical empirical challenge is that other tools target different languages [12], require annotation [15], or code migration [29]. One way to address this threat would be to compile a benchmark of unit inconsistencies, but that is beyond the scope of this effort.

6 RELATED WORK

In this section we continue a discussion of related work started in Section 1, where we identified several kinds of approaches for unit inconsistency detection.

Since the late 1970s, researchers have proposed programming language extensions and tool support to enable unit consistency checks, such as work by Gehani [11], who proposed extending Pascal, Hilfinger's Ada package [13], Wand and O'Keefe's simply-typed lambda calculus [34], Novak's work with unit conversion [23], Delft's extension for Java [33], Roşu's and Feng dynamic approach in C [28], Umrigar's compiler [32], Antoniu's spreadsheet checker XeLda [3], Jiang and Su's unit annotations in Osprey [15], and Schabel and Watanabe's `boost::units` for C++ [29]. More recently, unit consistency as envisioned by Kennedy [17] has been built into `F#`. This implementation seems to fully realize unit consistency, but does not appear to have been widely adopted[4]. Like this work, we detect inconsistent units during addition and assignment, but unlike this work our approach works without requiring extra annotations which often lower the barriers for tool adoption. Like these efforts we are concerned with unit consistency, but unlike these systems we are not proposing language extensions but rather seeking to leverage one mapping effort to enable unit inconsistency detection without requiring developers to modify their programs.

One of the more similar efforts is the tool UniFi [12] that infers dimensions automatically by mining a program for contradictory variable type usage, much in the same manner as Lackwit [24], but applied to unit inconsistency detection. Like those tools, we propagate an abstract type through assignment and detect inconsistent usage. Unlike their work we apply abstract types from outside the program and can detect inconsistencies without requiring the program to contain contradictory variable type usage. For example, our tool can detect the addition of inconsistent units in Figure 4 even if these variables were used only once in this program, whereas UniFi would not detect this inconsistency.

[4]Assessing levels and rates of tool adoption is difficult in a large and diverse community. However, we note that not one system among the 213 we explored uses any programming languages with unit support like `F#`, and only 3 use the `boost:units` library extension requiring annotation.

7 CONCLUSIONS

In this paper we presented a novel approach to detect physical unit inconsistencies in programs including cyber-physical, scientific computing, and embedded systems. The approach detects useful inconsistencies without modifying the program or shared libraries with annotations.

The approach is unique in that the mapping from physical attributes to units enables unit inconsistency detection without annotation, and also unique in that it favors speed over precision while still detecting meaningful unit inconsistencies. We implemented the approach in a prototype tool and evaluated it on a corpus of 213 systems, detecting inconsistencies in more than 11% of the systems. The tool has an 87% true positive rate for a class of inconsistencies it can detect with high confidence, and a preliminary survey of developers analyzing the tool's findings provides encouragement for further development and study.

In the future, we will pursue several research avenues. First, with the given approach and toolset in place, we are well positioned to detect many more types of unit inconsistency. One kind of inconsistencies that we have started to examine consists of those where the resulting units are not defined in SI. For example, in one of the systems we analyzed we found resulting units like `(meters, 1.5)` which does not have a real physical meaning. Although having such units seems odd, we need to investigate whether they warrant enough attention to take action. Second, as is common with this class of static analysis approaches, we will attempt to optimize the soundness, precision, and performance tradeoffs. In particular, we will explore improving path sensitivity, interprocedural analysis, and extending the scope of analysis since we have seen cases where they could reduce the number of false positives and move some of the findings from low to high confidence. This may require us to integrate the approach with richer analysis frameworks. Third, we intend to analyze the significance of these kinds of inconsistencies: their impact on systems at runtime, how much developer time is consumed with alternate methods such as manual unit annotations including constants, and the root causes of these inconsistencies. Fourth, we will extend the tool to other common cyber-physical libraries by introducing additional mappings, which will let us evaluate a larger number of systems. From an empirical perspective, we will also perform a more extensive survey of the tool findings, and submit more of the findings to developers' for evaluation. Last, we would like to move beyond the detection of inconsistencies, to providing recommendations on how to fix the unit inconsistencies.

ACKNOWLEDGEMENTS

We deeply appreciate the time and input from members of The Robotics Institute at Carnegie Mellon University, The Correll Lab at the University of Colorado, The Dunbabin Lab at Queensland University of Technology, the NIMBUS lab at the University of Nebraska-Lincoln, the Autonomous Space Robotics Laboratory at the University of Toronto, and the Robotics Algorithms & Autonomous Systems Lab at Virginia Tech. the RAAS Lab at Virginia Tech. This work was supported in part by NSF awards #1638099 and #1526652, USDA-NIFA #2013-67021-20947, and USDA-NIFA #2017-67021-25924

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, and others. 2005. The Fortress language specification. *Sun Microsystems* 139 (2005), 140.

[2] 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.

[3] Tudor Antoniu, Paul A Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. 2004. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 439–448.

[4] 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.

[5] 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.

[6] Percy Williams Bridgman. 1922. *Dimensional Analysis*. Yale University Press.

[7] 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.

[8] Satish Chandra and Thomas Reps. 1999. Physical type checking for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 24. ACM, 66–75.

[9] Al Danial. 2016. Count Lines Of Code. (2016). https://github.com/AlDanial/cloc

[10] Open Source Robotics Foundation. 2010 (accessed 27 July 2016). *ROS Enhancement Proposal 103*. http://www.ros.org/reps/rep-0103.html

[11] Narain Gehani. 1977. Units of measure as a data attribute. *Computer Languages* 2, 3 (1977), 93–111.

[12] Sudheendra Hangal and Monica S Lam. 2009. Automatic dimension inference and checking for object-oriented programs. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 155–165.

[13] Paul N Hilfinger. 1988. An Ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 189–203.

[14] David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 31. ACM, 13–19.

[15] Lingxiao Jiang and Zhendong Su. 2006. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *Proceedings of the 28th international conference on Software engineering*. ACM, 262–271.

[16] Michael Karr and David B Loveman III. 1978. Incorporation of units into programming languages. *Commun. ACM* 21, 5 (1978), 385–391.

[17] Andrew Kennedy. 1996. *Programming languages and dimensions*. Number 391. PhD Thesis, University of Cambridge.

[18] Andrew Kennedy. 2010. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*. Springer, 268–305.

[19] 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.

[20] Daniel Marjamäki. 2013 (accessed 1 February 2017). *Cppcheck: a tool for static C/C++ code analysis*. http://cppcheck.sourceforge.net/

[21] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. 2006. YARP: Yet another robot platform. *International Journal of Advanced Robotic Systems* 3, 1 (2006), 8.

[22] IM Mills, Barry N Taylor, and AJ Thor. 2001. Definitions of the units radian, neper, bel and decibel. *Metrologia* 38, 4 (2001), 353.

[23] Gordon S. Novak. 1995. Conversion of units of measurement. *IEEE Transactions on Software Engineering* 21, 8 (1995), 651–661.

[24] Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *In Proceedings of the 19th International Conference on Software Engineering*. Citeseer.

[25] International Bureau of Weights, Measures, Barry N Taylor, and Ambler Thompson. 2001. The international system of units (SI). (2001).

[26] Open Source Robotic Foundation. 2016. Automated Driving with ROS at BMW. (2016). http://www.osrfoundation.org/michael-aeberhard-bmw-automated-driving-with-ros-at-bmw

[27] ROS Industrial Consortium. 2016. Current Members - ROS Industrial. (2016). http://rosindustrial.org/ric/current-members

[28] 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.

[29] Matthias Christian Schabel and Steven Watanabe. 2008. Boost. *Units* 1, 0 (2008), 2003–2010.

[30] SoftBank. 2016. Romeo, the research robot from Aldebaran. (2016). http://projetromeo.com

[31] 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).

[32] Zerksis D Umrigar. 1994. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices* 29, 9 (1994), 135–139.

[33] André Van Delft. 1999. A Java extension with support for dimensions. *Software Prac. Experience* 29, 7 (1999), 605–616.

[34] Mitchell Wand and Patrick O'Keefe. 1991. Automatic Dimensional Inference.. In *Computational Logic-Essays in Honor of Alan Robinson*. 479–483.