Detecting Physical Unit Inconsistencies in Code

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

***Abstract*—Systems interacting with the physical world operate with physical units. When unit operations in the code are inconsistent with the physical units rules, those systems may suffer. Existing approaches to support unit consistency in code 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 code annotation, modification, or migration. It does so by capitalizing on existing libraries that handle standardized physical units, common in the robotics domain, to link program variables to physical units. Then, leveraging rules from dimensional analysis, the approach propagates and infers units through code, and detects inconsistent unit usage. We implement and evaluate the approach in a tool, analyzing 213 systems, 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 robotic system developers finds that the unit inconsistencies detected by the tool are 'problematic'.**

***Keywords*-physical units; program analysis; dimensional analysis; type checking; robot systems.**

I. Introduction

Systems that interact with the physical world operate over quantities measured in physical units. Consider the humanoid robot pictured in Figure 1. This robot perpetually perceives the world through depth sensors, cameras, and gyroscopes, and interacts with the world through its actions. The system collects measurements as it senses and acts, and transforms them into distances (`meters`) and angles (`radians`). The system also integrates these measures with others such as time (`seconds`) to estimate derived measures, for instance, 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 code snippet in Figure 2 belonging to the same robot and aimed at testing the control of high-level actions. The expression on lines 191-193 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 instead adds `meters` to `meters-squared`. The code compiles without complaint as both variables have the same programming type. Yet the inconsistency in how the physical units are combined in the code constitutes a fault that will go undetected by the type system, likely to manifest later as incorrect behavior.

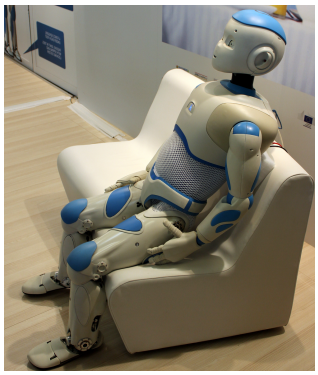

Fig. 1: Humanoid Romeo [1].

The consequences of such unit inconsistencies in systems interacting with the physical world exhibit a range of severities, from mild to occasionally catastrophic [2]. There does not seem to exist, however, an authoritative estimate of how frequently physical 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 advocated for the design of programming languages with support for physical unit types as well as the abstraction of physical units into dimensions [3], now realized in languages like `F#` [4] and `fortress` [5], domain-specific languages [6] or libraries [7], and specialized annotations [8], [9], [10]. Some solutions have been deemed effective in practice [11], but all require system developers to incur additional up-front migration and development costs, hindering widespread adoption.

In this work we propose a different approach to finding unit inconsistencies that more closely mirrors those found in many popular bug finding tools [12]. Our tool sacrifices soundness and completeness to perform fast and scalable detection without placing any code annotation, modification, or migration burden on developers. 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.

Our approach bridges the gap between physical units and code. As shown in Figure 3, our approach maps the physical libraries that handle physical quantities to standardized physical units. It also defines dimensional analysis rules that govern how these units can be manipulated as part of code expressions. As our approach analyzes the code of a target system, the library unit mapping enables the automatic decoration of variables with units, and the unit rules enable the propagation and inference of units across the code. Then, the approach traverses the code representation to detect locations where units are assigned, compared, or combined inconsistently.

```
189 float computeDistance(geometry_msgs::Pose goal, geometry_msgs::Pose current)
190 {                                                                  meters-squared
191   float dist = (goal.position.x - current.position.x)*(goal.position.x - current.position.x)
192       + (goal.position.y - current.position.y) + (goal.position.y - current.position.y)
193       + (goal.position.z - current.position.z) + (goal.position.z - current.position.z);
194                                                          meters
195   return dist;
196 }
```

Fig. 2: Code snippet with 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

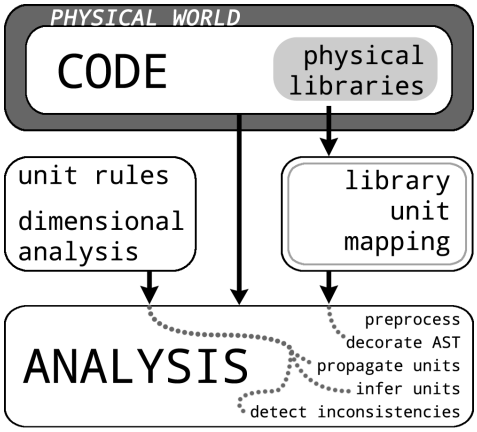

Fig. 3: High-Level Approach.

```
463                meters-per-second          meters
464   //pass along drive commands
465   cmd_vel.linear.x = drive_cmds.getOrigin().getX();
466   cmd_vel.linear.y = drive_cmds.getOrigin().getY();
```

Fig. 4: Inconsistent assignment.
*package:*ros-planning *source:*https://git.io/v6XlV

```
65  if (fabs(twist.linear.y) > fabs(twist.angular.z))
66  { meters per second   radians per second
67    marker_.points[1].y = twist.linear.y;
68  } else {
69    marker_.points[1].y = twist.angular.z;
70  }
```

Fig. 5: Inconsistent comparison.
*package:*ros-teleop *source:*https://git.io/v6Xld

The key contributions of this work are:

- A novel approach for detecting unit inconsistencies in existing code without extra effort from developers.
- A lightweight static analysis tool implementing the approach for systems built on the Robot Operating System (ROS) [13]. Our tool is available for download from http://nimbus.unl.edu/tools/.
- An initial evaluation of our tool on a corpus of 213 systems, wherein 24 systems (11%) contain unit inconsistencies, with a true positive rate of 87% for a class of unit inconsistencies we detect 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 developers of such systems.

II. Motivation

To provide a more concrete idea of the inconsistencies we aim to detect, we now examine three more code excerpts identified by our automated tool as having unit inconsistencies.

The code snippet shown in Figure 4 shows an assignment in line 465 with `cmd_vel.linear.x` receiving the value of `drive_cmds.getOrigin().getX()`. To make the code in the figure easier to understand, we have marked variables with their physical units. The `float64` data type is used for both `cmd_vel.linear.x` and `getX()`. However, `cmd_vel.linear` is part of a physical library called `ROS::geometry_msgs::Twist` with a library unit mapping to `meters-per-second`, while `getX()` on the right-hand-side of the assignment returns physical units of `meters`. The physical units actually assigned to the variable are incorrect with respect to the physical unit specification for that variable type. We call this kind of inconsistency *assignment of multiple units*. This code compiles without explicit warning while implicitly converting from one physical 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 robot behaviors.

A second code example is shown in Figure 5, where at line 65 system developers compare two variables' magnitudes. The underlying assumption is that the variables contain values that are *comparable*, that is, the physical units associated with the quantities should be the same. The comparison is between `twist.linear.y` and `twist.angular.z`, with units `meters-per-second` and `radians-per-second`, respectively. Therefore these two quantities are not comparable, 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 usually suspect, and should likely be well-documented, especially if intended for reuse.

A third example of unit inconsistency occurs entirely on the right-hand-side of statements during addition, as shown in Figure 6 on lines 1094-1097. This implements a distance function between three quantities: `force.x`, `force.y`, and `torque.z`. Specifically, this code squares each of the three

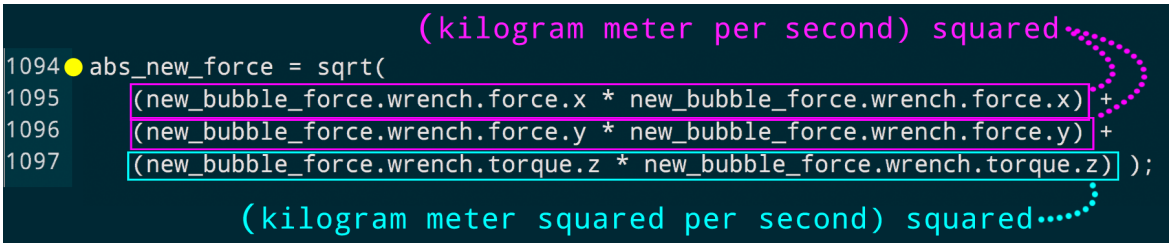

Fig. 6: Inconsistent units during addition of force and torque in distance metric.
full code at: https://git.io/v6X8T

quantities, then sums them and calculates the square root. The problem with this expression is that the physical units for `force` are different than the physical units for `torque`. Adding the square of `force` to the square of `torque` is unit inconsistent. We call this *addition of inconsistent units*. In the developer's defense, this calculation might behave as intended within a limited range of input values that effectively normalizes these values, but in general, adding quantities with dissimilar physical units yields a result devoid of physical meaning. Without physical meaning, the use of this distance metric might be considered a bewildering hack that works on one particular robot, 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 are a barrier to code re-use, since system developers must duplicate the robot and environment or risk unintended behavior.

These examples illustrate how having variables that can take different units, comparing inconsistent units, and adding inconsistent units can impact code comprehension, correctness, and reuse. Our study findings in Section V provides further evidence of the problems associated with these unit inconsistencies, and shows that they lurk in at least 11% of the 213 systems we analyzed.

Now that we have shown some examples of unit inconsistencies, we present the approach.

III. Approach

We now describe the approach, including the flow of the analysis explained with examples and pseudocode, and begin with a summary of the guiding requirements that inform the approach.

A. Requirements

The goal is to provide fast, 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[1].

[1]Both Bessey and Hovemeyer *et al.* used $< 20\%$ as a baseline [12], [14]

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.

B. Library Unit Mapping and Unit Rules

At a high-level, the proposed approach works as illustrated in Figure 3. Our approach integrates physical units and dimensional analysis with the code and the physical libraries it uses. Critical to that integration is the mapping we provide for associating the physical libraries to their meaning in the physical world, and the application of rules that dictate how to manipulate variables representing physical units. We start by discussing this mapping and rules that serve the program static analysis.

Mapping Library Units to Physical Units. Physical phenomena are quantified in terms of measured physical units, such as `meters-per-second` or `furlongs-per-fortnight`. The International System of Units [15] defines the standard SI system, where units are either *derived* or *base*. The base units are:

$$\mathsf{SI} = \{\text{kilogram, meter, second, ampere, kelvin, mole, candela}\} \quad (1)$$

These seven base units can be used in combination to express all 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, one `kilogram-meter-per-second-squared`. Our approach targets the set of SI base units from Equation 1 and we also include the derived `radian` unit, a dimensionless `meter-per-meter` used to measure angles because angular position and velocity are ubiquitous in robotic systems. Using these units, we define the set of all possible combinations of units with integer exponents:

$$2^{\mathsf{SI}} = \mathbb{P}\{\mathsf{SI} \times \mathbb{Z}\}$$

The elements of the powerset 2^{SI} include all familiar derived units like velocity {(`meters`, 1), (`seconds`, -1)}, or equivalently `meters-per-second`. We consider only the simplest forms of units where identical bases are combined, and base units with exponent 0 (zero) become 1 (one) and are elided.

Given the set 2^{SI}, we next define the relationship

$$\mathsf{R}_{\mathsf{LIB\_UNITS}} : \mathsf{LIB} \rightarrow 2^{\mathsf{SI}} \tag{2}$$

This relation maps from the set of libraries operating with physical units (physical libraries) LIB to the set of possible physical unit assignments 2^{SI}.

As mentioned, this work is instantiated in the context of robotic systems, which often use libraries to work with standardized sensors and actuators that interact with the physical world using physical units. In this work our approach targets one such standard physical library, the one belonging to the Robot Operating System (ROS) [13]. ROS is a popular middleware to enable rapid development in robotic systems widely adopted by academic and professional developers, including industrial automation at Boeing [16] and a high-profile autonomous driving effort at BMW [17].

In ROS libraries, physical classes are pre-defined to represent specific physical units. For example, the physical class `ROS::geometry_msgs::Twist` *"expresses velocity in free space broken into its linear and angular parts"*, [18]. The layout of this data structure and its mapped units are:

- `twist.linear.x` (float64): `meters-per-second`
- `twist.linear.y` (float64): `meters-per-second`
- `twist.linear.z` (float64): `meters-per-second`
- `twist.angular.x` (float64): `radians-per-second`
- `twist.angular.y` (float64): `radians-per-second`
- `twist.angular.z` (float64): `radians-per-second`

Beyond this example, ROS adheres to SI as articulated in ROS REP-103 ('ROS Enhancement Proposal') [19] for all physical units. The process we followed to generate the mapping for ROS consisted of examining the documentation and when in doubt the code for the ROS physical classes and determining the unit mapping for each class attribute. We also monitored the top classes invoked across the systems we studied, identifying a few more classes that were not in the standard code location of the ROS physical library. As of the latest instantiation, our prototype includes 82 classes of units (e.g. `nav_msgs::Odemetry`, `geometry_msgs::Inertia`, `sensor_msgs::FluidPressure`) with 246 total attributes, mapping to 17 distinct derived units $\in 2^{\mathsf{SI}}$.

Unit Rules. We have defined a set of rules R that dictate how the physical units are combined as they appear in mathematical expressions. Similar to other unit systems, these rules provide a basic *equational theory* [4], [20]. For example, such rules convey that multiplying `meters-per-second * per-second` yields `meters-per-second-squared`, and multiplying a term by its inverse, such as `meter * per-meter` yields 1 (one). Rules capture the operational semantics of the operators through the form of $\frac{premise}{conclusion}$ and mimic much of the work in dimensional analysis [21], which abstracts units to dimensions [22]. For example, two equations for the moment of inertia, one expressed in `kilogram-meters-squared` and the other as `pound-feet-squared` use different units but are dimensionally equivalent, both expressing `mass-length-squared`. Although our rules operate at the dimension level, the approach retains the units during the analysis to include them in any inconsistency message generated by the approach to make them more intuitive to system developers. Examples for the multiplication and addition rules are provided in Equations 3 and 4 respectively.

Our approach includes similar rules for other operators that are commonly involved in the manipulation of physical variables in code, including addition, subtraction, multiplication, division, square root, exponents, min, max, absolute value, floor, ceiling, and the ternary operator. We note that although $\mathsf{R}_{\mathsf{LIB\_UNITS}}$ depends on the library used, the unit rules are likely to remain the same across systems and libraries.

C. Analyzing for Unit Inconsistencies

Our approach leverages these mappings and rules to detect unit inconsistencies through a static analysis consisting of five phases: preprocessing, unit decoration of the abstract syntax tree (AST), unit propagation through the AST, unit inference for variables, and unit inconsistency detection. We now describe each phase in more detail, and refer to Algorithm 1 for a full description.

Preprocessing. The approach begins by pre-processing program P into an ordered list of functions. It constructs the call graph and performs a reverse topological sort, so that it can analyze functions before they are called. If the topological sort yields a partial order, the approach breaks ties arbitrarily and examines only the first ordering for simplicity. This preprocessing is shown in Algorithm 1 lines 11-12.

Unit Decoration of AST Leaves. Each function $f \in \mathsf{P}$ is an ordered list of n statements $< statement_1, statement_2, \cdots, statement_n >$. Each $statement_i$ represents an AST graph (V, E), with the vertex set denoted $statement_i.V$. Vertices in $statement_i$ consist of the elements of the language of the program and include the program variables, VAR.

This phase decorates variables in leaves of the AST with physical unit labels, using two relations. The first relation links program variables to physical classes in physical libraries through $\mathsf{R}_{\mathsf{VAR\_LIB}}$:

$$\mathsf{R}_{\mathsf{VAR\_LIB}} : \mathsf{VAR} \rightarrow \mathsf{LIB}$$

This relation maps from the set of program variables VAR to the set of physical program libraries LIB, by analyzing the variable declarations. The second relation links physical libraries to physical units using $\mathsf{R}_{\mathsf{LIB\_UNITS}}$ (Equation 2).

We can see how these relations are used using an example shown in Figure 7. This figure shows the AST for the code statement in Figure 6. This code computes a distance metric by summing the squares of three quantities, then taking the square root. In Figure 7 towards the bottom-left of the AST, the variable `force.x` is a $var \in$ VAR and the relation $\mathsf{R}_{\mathsf{VAR\_LIB}}$ links `force.x` to physical class `ROS::wrench.force` $\in$ LIB. The physical class `ROS::wrench.force` $\in$ LIB is *pre-defined* to have units $\{(\texttt{kilogram}, 1), (\texttt{meters}, 1), (\texttt{second}, -1)\} \in 2^{\mathsf{SI}}$, abbreviated `kg m` s^{-1} in the figure. The relation $\mathsf{R}_{\mathsf{LIB\_UNITS}}$

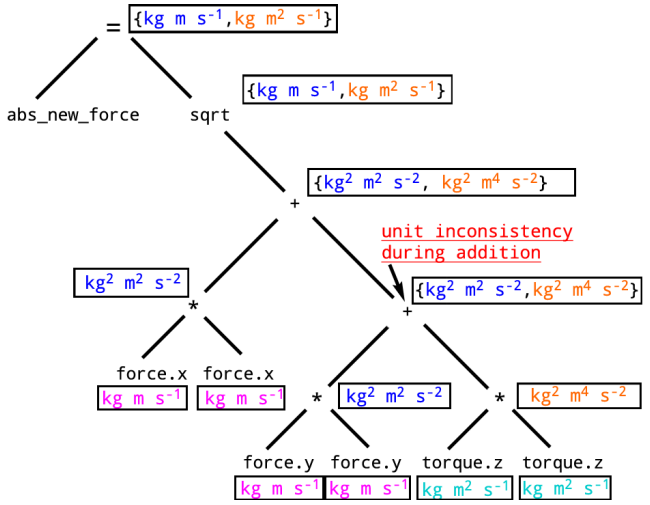

Fig. 7: Example AST of statement from Figure 6 with `new_bubble_force.wrench` omitted for simplicity. Figure shows unit decoration of variables, propagation toward the root by rules, and unit inconsistency detection when trying to add `force-squared` to `torque-squared`.

connects the variable `force.x` to the unit decoration `kg m` s^{-1}. In this example, `force.x` is decorated with physical units because of its physical class.

Variables are decorated with physical units in two cases: 1) their physical library has specified units; or 2) they were previously assigned physical units. In this latter case, the physical units will be in the relation:

$$\mathsf{R}_{\mathsf{SYMBOL\_TABLE}} : \mathsf{VAR} \rightarrow 2^{\mathsf{SI}}$$

This relation represents the assignment of program variables VAR to a specific unit assignment, and we will discuss assignment statements in more detail shortly. If a $var \in \mathsf{VAR}$ does not belong to a physical class and has not had a prior assignment, it remains undecorated.

The AST decoration phase is shown in pseudocode in Algorithm 1 on lines 14-18. Notice that AST decoration happens only once for each vertex in a statement's AST Graph, $statement.V$. The approach only moves forward in a linear flow-insensitive pass to keep the analysis light and simple.

Unit Propagation in the AST. Unit propagation is the process within the AST whereby unit decorations at the leaves move and combine up the tree toward the root. During unit propagation the approach examines every vertex of the AST and tries to apply local rules that allow unit decorations to propagate 'up' to the current vertex by examining the unit decorations immediately 'below' it. By repeatedly visiting the AST and applying these unit rules, our approach propagates unit decorations from the leaves to the AST root.

Continuing with our example from Figure 7, imagine the AST has only been decorated with units at the leaves (such as the variable `force.x` which was decorated with `kg m` s^{-1}). Since there is a multiplication symbol '$*$' that has children with unit decorations, the unit propagation rule for multiplication (and division) applies:

$$\frac{\text{`a}\{*,\div\}\text{b'}, \quad \mathsf{a_u} \leftarrow \mathsf{R_{units}}(\mathsf{a}), \quad \mathsf{b_u} \leftarrow \mathsf{R_{units}}(\mathsf{b})}{\{\mathsf{R}_{*=}(\alpha,\beta) | \alpha \in \mathsf{a_u}, \beta \in \mathsf{b_u}\} \in 2^{\mathsf{SI}}} \tag{3}$$

This rule states that if a vertex is a multiplication symbol '$*$' and its children a and b have unit decorations, then the '$*$' symbol is decorated with the result of multiplying the units of a with the units of b. Unit multiplication of a and b proceeds according to the equational theory rule $\mathsf{R}_{*=}$ (Kennedy [4]) for multiplying units. This equational theory is made of straightforward rules, like `meters` $*$ `meters` equals `meters-squared`. In the example from Figure 7, the new units for `force.x` $*$ `force.x` are new units `kg`^{2}`m`^{2}`s`$^{-2}$. These new units then become a new unit decoration for the symbol '$*$', moving up the AST from the leaves.

Continuing with the example, the units are then propagated up from the multiplications to the addition vertices, according to the addition propagation rule:

$$\frac{\text{`a}\{+,-\}\text{b'}, \quad \mathsf{a,b} \in V, \quad \mathsf{a_u} \leftarrow \mathsf{R_{units}}(\mathsf{a}), \quad \mathsf{b_u} \leftarrow \mathsf{R_{units}}(\mathsf{b})}{\mathsf{a_u} \cup \mathsf{b_u}} \tag{4}$$

This rule specifies that the result of adding two units is the set union of the units of the operands. In Figure 7, this results in a set of multiple possible units, because `force-squared` is added to `torque-squared`, yielding a unit decoration of $\{$`kg`^{2}`m`^{2}`s`$^{-2}$, `kg`^{2}`m`^{4}`s`$^{-2}\}$. Tracking the *assignment of multiple units* for a variable may help us detect inconsistencies when they are later used. Then, the units propagate up to the '`sqrt`' vertex, which propagates similarly to multiplication propagation in Equation 3, but instead of unit exponent addition, we apply exponent division by two, resulting in $\{$`kg m` s^{-1}, `kg` $m^2 s^{-1}\}$. Finally, these units propagate up to the assignment statement '=', and since this is the root of the AST, no further propagation is possible and the AST reaches *unit quiescence*. During AST unit propagation, unit quiescence is guaranteed because unit decorations can only propagate up the tree toward the root and the tree is finite. Unit quiescence occurs at the end of unit propagation in the AST, and this is the only time in our analysis that decorations are added to statements.

This process of unit propagation in the AST is also shown in Algorithm 1 lines 20-29. The algorithm continues to visit vertices in the AST, applying the proper propagation rules depending on the operator, as long as the unit decoration for any vertex changes. When no decorations change, the AST has achieved unit quiescence, and the unit propagation on this AST stops.

If at any point during AST exploration an unknown symbol is encountered at a vertex, such as an external function call, then ISOPERATOR returns false in line 25 and the vertex remains unlabeled and effectively blocks unit propagation up the tree from that vertex. Vertices might also remain undecorated because the leaf contains either a constant or unknown variable (in neither $\mathsf{R}_{\mathsf{VAR\_LIB}}$ nor $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$).

Undecorated vertices in the AST are treated as 'unknowns' in our analysis and can lessen our confidence in the unit analysis.

Our unit propagation rules also includes a 'confidence' decoration per vertex in the AST. Our approach defaults to 'high-confidence'. It only assigns 'low-confidence' when applying the vertex operator on unknown units could result in any other unit assignment in 2^{SI} or an unknown variable (with no link in $\mathsf{R}_{\mathsf{VAR\_LIB}}$). For example, as defined by dimensional analysis, multiplying by unknown units could result in new units or even a constant, so multiplication with unknown units always receives a low-confidence designation. We omit the 'confidence' label from our figures for simplicity.

Unit Inference During Assignment and `Return`. After unit decoration and propagation, the approach infers physical units for variables and functions from assignment and return statements at the root of the AST. Assignment and return statements add to the relation $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$, allowing subsequent references to variables and functions to be decorated with previously inferred units. Indeed, the only way the approach gains knowledge for use in later statements is through assignment and returns. This is why it is important to sort the functions during preprocessing, so more units can be available at call points, if possible. The approach preserves previous unit assignments for consideration during inconsistency detection, and applies the most recent units during decoration.

In the example in Figure 7, the relation $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$ is updated to contain a mapping from `abs_new_force` to two potential units, $\{\texttt{kg m s}^{-1}, \texttt{kg m}^2\texttt{s}^{-1}\}$, the unit decoration at the root of the AST. This variable decoration is then available to subsequent statements seeking the physical units assignment to `abs_new_force`. Notice this unit assignment reflects two different units resulting from the addition, a fact that becomes useful during inconsistency detection.

Unit inference is shown in Algorithm 1 on lines 31-38. When a statement is an assignment or return, then the unit decoration at the root of the AST contains the culmination of the unit propagation rules. The unit decoration in the root of the AST is then transferred to the relation $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$, where it can be read later. Once all statements in all functions have been decorated, and all units propagation rules have quiesced, then unit inconsistency detection begins.

Unit Inconsistency Detection. This phase examines two objects: the symbol table relation $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$ and decorated statement ASTs.

Scanning $\mathsf{R}_{\mathsf{SYMBOL\_TABLE}}$ can reveal two inconsistency patterns: 1) variables that are assigned units contrary to their physical library specification; and 2) variables assigned different units at different points in the program.

Examining each fully decorated statement AST in the program can reveal two other inconsistency patterns: 1) addition (or substraction) of inconsistent units, which can happen in any kind of expression, not just assignment; and 2) comparison of inconsistent units, which can happen in expressions.

The flow of the unit inconsistency detection phase is shown in Algorithm 1 in lines 40-50. The approach makes a separate pass over the whole program P, function by function. Once

Algorithm 1 Physical unit inconsistency detection over P

```
1: function DetectUnitInconsistency(P)
2:     R_SYMBOL_TABLE ← ∅
3:     sortedFunctions ← Preprocess(P)
4:     for function ∈ sortedFunctions do
5:         for statement ∈ function do
6:             DecorateWithUnits(statement)
7:             PropagateUnits(statement)
8:             InferUnits(statement)
9:     return DetectInconsistencies(P)

10: function Preprocess(P)
11:     callGraph ← ConstructCallGraph(P)
12:     return TopoSortReverse(callGraph)

13: function DecorateWithUnits(statement)
14:     for v ∈ statement.V do
15:         if isVariable(v) then
16:             v.units ← R_LIB_UNITS(R_VAR_LIB(v))
17:             if R_SYMBOL_TABLE(v) ≠ ∅ then
18:                 v.units ← v.units ∪ R_SYMBOL_TABLE(v)

19: function PropagateUnits(statement)
20:     unitQuiescence ← False
21:     while ¬ unitQuiescence do
22:         unitQuiescence ← True
23:         for v ∈ statement.V do
24:             units′ ← ∅
25:             if isOperator(v) then
26:                 units′ ← ApplyPropagationRules(v)
27:                 if v.units ≠ units′ then
28:                     v.units ← v.units ∪ units′
29:                     unitQuiescence ← False

30: function InferUnits(statement)
31:     if isAssignment(statement) then
32:         units′ ← statement.V.root.units
33:         if units′ ≠ ∅ then
34:             R_SYMBOL_TABLE(V.root.lhs) ← units′
35:     else if isReturn(statement) then
36:         units′ ← statement.V.root.units
37:         if units′ ≠ ∅ then
38:             R_SYMBOL_TABLE(function) ← units′

39: function DetectInconsistencies(P)
40:     UI ← ∅                                  ▷ Unit Inconsistencies
41:     for var, units ∈ R_SYMBOL_TABLE do
42:         if |units| > 1 then
43:             UI ← UI ∪ {(var, units)}        ▷ Multiple Units
44:     for function ∈ P do
45:         for statement ∈ function do
46:             for v ∈ statement.V do
47:                 if isAdditionOrComparison(v) then
48:                     if v.left.units ≠ v.right.units then
49:                         UI ← UI ∪ {(v, v.units)}
50:     return UI
```

all the unit inconsistency checks have been completed, the approach yields a set of unit inconsistency messages indicating inconsistency type, confidence level, line number, variables involved and physical units present in the statement.

The example in Figure 7 shows unit inconsistency detection during addition. As every vertex in the AST is visited, this rule recognizes the addition symbol '+' with child vertices (the operands) having different units. As shown in the figure, this mechanism detects the inconsistent addition of $\texttt{kg}^2\texttt{m}^2\texttt{s}^{-2}$ to $\texttt{kg}^2\texttt{m}^4\texttt{s}^{-2}$ (the `meters` exponents are different). An example inconsistency message from this example reads: `Addition of inconsistent units on line 1094 with high-confidence. Attempting to add` $\texttt{kg}^2\texttt{m}^2\texttt{s}^{-2}$ `to` $\texttt{kg}^2\texttt{m}^4\texttt{s}^{-2}$.

Limitations. By design, the approach is meant to be unsound and incomplete, but fast and effective at detecting meaningful unit inconsistencies. As described, the approach is flow insensitive and intra-procedural, only supports the mapped libraries, and does not handle complex constructs like pointers or STL data structures. 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. We also realize that the kind of unit inconsistencies that can be currently detected with the approach are limited and represent just a part of what is possible. Extending the approach to detect other inconsistencies like those presented by units that lack meaning in the physical world is among our next steps.

IV. Prototype Implementation

Our implementation architecture closely follows the approach and is implemented in 3300 lines of python code, and can be invoked as a command-line tool. We also use several other tools to facilitate parsing and code analysis.

Our tool currently utilizes CPPCheck as a `C++` preprocessor and parser [23]. It invokes CPPCheck 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 configuration. Our tool also uses the CPPCheck python library for parsing the dump files, and we have enriched some of the existing classes to store the tree decoration. We use the open-source NetworkX [24] graph libraries for building a function call graph and topologically sorting it.

We use a visitor pattern to implement our AST operations, decorations with $\mathsf{R}_{\mathsf{VAR\_LIB}}$, and propagations with unit rules. We recursively walk the AST generated by CPPCheck, and implement the relation $\mathsf{R}_{\mathsf{LIB\_UNITS}}$ using a python dictionary-of-dictionaries. During implementation we realized that several physical units require specialized handling. For the physical library `ROS::sensor_msgs::JointState` we assume that `velocity` and `position` are angular instead of linear since this appears to be the most common case. Given this ambiguity, we mark every JoinState decoration as low-confidence. `Radian` is handled differently depending on the expression. In multiplication expressions, `Radian` acts as a dimensionless scalar since its units in the SI system are `meters-per-meter`, and therefore its units act like a '1'. In addition, it behaves as a 'coherent units of measure' [25], meaning that it cannot be added to a dissimilar unit, even though it is dimensionless. Still further from intuition is the dimensionless `quaternion`, employed widely in robotic systems to describe 3-dimensional rotations. Both the quaternion and the radian are closed under multiplication, so `Radian-squared` resolves to `Radian`.

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

TABLE I: ROS Open-Source Repositories

Although primarily a command-line tool, we optionally write unit inconsistency results to a SQL database and built an austere HTML interface to explore unit inconsistency messages organized by type and repository.

Visit http://nimbus.unl.edu/tools to download our tool.

V. Evaluation

The goal of the evaluation is to assess the ability of the tool to detect unit inconsistencies that may cause problems for real robotic 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 unit inconsistencies detected by our tool as problematic.

A. 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, is organized by the version name of the core ROS libraries, with the latest long-term release at the time of this work being 'Indigo Igloo.' The Indigo repositories include 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. Some of these diverse projects utilize ROS message libraries with specified physical units (called 'physical libraries' in this paper).

Table I shows statistics about this corpus. At the time we gathered this corpus of ROS code, there were 2416 projects

linked from the 'Indigo' version of ROS. Of these 2416 links, 649 were linked to live github 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 ROS physical libraries by text searching for the names of any ROS libraries with physical unit associations, like "geometry_msgs::Twist." For this initial work, we proceed with all ROS `geometry`, `navigation`, `transform`, `sensor`, and `time` libraries.

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 [26]. 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 with a 2.9 GHz Intel Core i5 processor, and 16GB of memory.

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). 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: 1) a variable is or can be assigned multiple units; 2) a variable is or can be assigned physical units that disagree with the units specified by its physical class definition (its ROS message type); 3) the addition of two variables with inconsistent units.

Our results are summarized in Table II. The overall TP rate, computed as $TP\% = 100 * TP/(TP + FP)$, for 'high-confidence' physical unit inconsistencies is 87.0%. This includes the three types, variables assigned multiple units, addition of inconsistent units, and comparison of incompatible units. As we noted earlier, for one of the cases where we contacted the authors of the code for clarification, shown in Figure 2, 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 vector cross-products and outer-products, where `meters-squared` *intentionally* equals `meters`. Figure 8 contains one such case in line 90, which is frequently used and deemed correct by roboticists. 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.

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 primary culprit for the relatively low TP rate is *incomplete variable unit information*. These are often variables with an implicit physical unit but without an associated physical class for the tool to identify the units. We noticed that often these variables have names

```
86 inline geometry_msgs::Point32
87   cross (const geometry_msgs::Point32 &p1,
           const geometry_msgs::Point32 &p2)
88 {  meters                    meters-squared
89   geometry_msgs::Point32 r;
90   r.x = p1.y * p2.z - p1.z * p2.y;
91   r.y = p1.z * p2.x - p1.x * p2.z;
92   r.z = p1.x * p2.y - p1.y * p2.x;
93   return (r);
```

Fig. 8: False positive during cross-product.
*package:*pr2-navigation *source:*https://git.io/v6xS7

```
104  m_thrust += 10000 * dt;
105  geometry_msgs::Twist msg;          seconds
106  msg.linear.z = m_thrust;
     meters-per-second
```

Fig. 9: False positive with constant.
*package:*crazyflie_ros *source:*https://git.io/v6x7T

like '`dT`' for change in time or '`vx`' for linear velocity in x, names that imply physical units, and could be used as part of heuristics to reduce the FPs. Another source of FPs are constants, including hard-coded parameters never factored out into named variables. For example, Figure 9 shows an assignment in line 106 to `msg.linear.z` from `m_thrust`, where `m_thrust` has been assigned units from a previous assignment based on an addition with the constant 10000 times `seconds`. Since the units for `msg.linear.z` are `meters-per-second` and it is assigned units of `seconds`, our tool reports an inconsistency. Since the units for 10000 are unknown, the unit inconsistency reported on line 106 is reported as 'low-confidence.' In this case the code could be correct if the unit for the constant was ms^{-2}.

B. Survey

We conducted a survey to obtain an initial assessment of whether these kinds of unit inconsistencies are problematic to robotic software developers. Our survey instrument consists of eight questions, each showing a code example similar to those in Figures 4-6, 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 'yes-maybe-no' question the respondents had a chance to explain their answer. The order of the eight questions was randomized for each respondent.

The target population for our survey was either heads of academic robotics research labs or their senior research associates. These labs publish regularly in top robotic conferences and they 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 therefore may not generalize, but at this stage of the work we wanted quick,

	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 II: Classification of unit inconsistencies found by our tool.

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 4)	2	8	0
7 (Figure 6)	7	3	0
8	4	5	1
TOTALS	45	27	8
%	56%	34%	10%

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

initial feedback before attempting a larger, more nuanced study that might be justified in the future.

Our results are shown in detail in Table III. 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. Only one respondent indicated pure uncertainty with *'I'm not sure.'*

Only 10% of responses reported 'no', *not* problematic. Four of the eight 'no's came from one respondent, who explains: *'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 III of the survey are also presented in this work as Figure 4 and 6. 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 6) 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 unit inconsistencies are problematic and that a tool to detect such inconsistencies is valuable to the robotic software community.

C. Threats to Validity

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. In at least one case our tool was wrong about *how* the units were inconsistent but right that they were inconsistent. Low-confidence FPs were more straightforward to identify and often were attributable to a single variable with an unknown unit decoration, such as the code example in Figure 9.

We mitigated the previous threat to internal validity by reviewing the results multiple times and discussing examples with other robotics 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.

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 robotic systems reg-

ularly incorporate sensors and actuators that utilize standardized middleware libraries shared across platforms, all using some convention for data structures containing physical units. Similarly, we recognize that we built R_{LIB_UNITS} for ROS, but other mappings to other middleware would help to generalize the approach. We are currently exploring other middleware, and will likely target the popular OROCOS robotic system [27] next.

VI. Related Work

Since the late 1970s, researchers have proposed programming language extensions and tool support to enable unit consistency checks, such as work by Gehani [28], who proposed extending Pascal to support unit consistency, Hilfinger's Ada package [8], Novak's work with unit conversion [29], Delft's extension for Java [30], Roşu's and Feng dynamic approach to unit checking in C [31], or Umrigar's compiler [9] and Schabel and Watanabe's `boost::units` [7] for C++. These works follow the general outline proposed by Karr and Loveman [3], who in 1978 outlined a general plan for supporting unit-consistency in programming languages. Like these efforts we are concerned with unit consistency, but unlike these systems we are not proposing language extensions but rather the incorporation of relations so that existing libraries handling physical units can be leveraged, together with dimensional rules, to enable the detection of unit inconsistencies with a particular focus on the robotics domain.

More recently, unit consistency as envisioned by Kennedy [21] has been built into `F#`. This implementation seems to fully realize unit consistency, but it does not appear to have been adopted by the robotic community. [2] Like this work, we detect inconsistent units during addition and assignment, but unlike this work our prototype tool works without requiring extra annotations which often lower the barriers for tool adoption.

Within the robotics domain, Biggs and MacDonald [32] presented a general method for implementing units-of-measure consistency in robotic programming environments, and Biggs extended this work into a prototype domain-specific programming language RADAR in his thesis [6]. Like their work we are focused in the robotic domain, but unlike their work our approach and tool is meant to work with existing code bases.

VII. Conclusions

In this paper we presented a novel approach to detect physical unit inconsistencies in code. The approach is fast, scalable, works with most common units in robotic systems, and does not impose additional burden on the developer in order to use it.

The approach is unique in how it integrates the mapping of code libraries to physical units and the dimensional analysis rules to expose unit inconsistencies. We implemented the approach in a prototype tool and evaluated it on a corpus of 213 systems built on top of the popular ROS middleware, 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 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 will extend the tool to other common robotic 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.

[2] 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 full physical unit support, and only 3 use the `boost:units` library extension for unit support which requires developers to annotate their code.

Acknowledgments

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 Autonomous Space Robotics Laboratory at the University of Toronto, and the Robotics Algorithms & Autonomous Systems Lab at Virginia Tech, and the NIMBUS Lab. This work was supported in part by NSF awards #1638099 and #1526652, and USDA-NIFA #2013-67021-20947.

References

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

[2] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. LaPiana, P. Rutledge, D. Folta, and R. Sackheim, "Mars climate orbiter mishap investigation board phase I report, 44 pp," *NASA, Washington, DC*, 1999.

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

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

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund *et al.*, "The fortress language specification," *Sun Microsystems*, vol. 139, p. 140, 2005.

[6] G. Biggs, "Designing an application-specific programming language for mobile robots," Ph.D. dissertation, ResearchSpace@ Auckland, 2007.

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

[8] P. N. Hilfinger, "An Ada package for dimensional analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 2, pp. 189–203, 1988.

[9] Z. D. Umrigar, "Fully static dimensional analysis with C++," *ACM SIGPLAN Notices*, vol. 29, no. 9, pp. 135–139, 1994.

[10] S. Chandra and T. Reps, "Physical type checking for C," in *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 5. ACM, 1999, pp. 66–75.

[11] T. Muranushi and R. A. Eisenberg, "Experience report: Type-checking polymorphic units for astrophysics research in Haskell," in *ACM SIGPLAN Notices*, vol. 49, no. 12. ACM, 2014, pp. 31–38.

[12] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

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

[14] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1. ACM, 2005, pp. 13–19.

[15] I. B. of Weights, Measures, B. N. Taylor, and A. Thompson, "The international system of units (SI)," 2001.

[16] ROS Industrial Consortium. (2016) Current members - ROS Industrial. [Online]. Available: http://rosindustrial.org/ric/current-members

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

[18] ——. (2010) geometry_msgs : Twist message. [Online]. Available: http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html

[19] O. S. R. Foundation, *ROS Enhancement Proposal 103*, Jul. 2010 (accessed 27 July 2016). [Online]. Available: http://www.ros.org/reps/rep-0103.html

[20] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.

[21] A. Kennedy, *Programming languages and dimensions*. University of Cambridge, Computer Laboratory, 1996, no. 391.

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

[23] D. Marjamäki, "Cppcheck: a tool for static C/C++ code analysis," 2013.

[24] D. A. Schult and P. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, vol. 2008, 2008, pp. 11–16.

[25] I. Mills, B. N. Taylor, and A. Thor, "Definitions of the units radian, neper, bel and decibel," *Metrologia*, vol. 38, no. 4, p. 353, 2001.

[26] A. Danial. (2016) Count Lines Of Code. [Online]. Available: https://github.com/AlDanial/cloc

[27] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3. IEEE, 2001, pp. 2523–2528.

[28] N. Gehani, "Units of measure as a data attribute," *Computer Languages*, vol. 2, no. 3, pp. 93–111, 1977.

[29] G. S. Novak, "Conversion of units of measurement," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 651–661, 1995.

[30] A. Van Delft, "A Java extension with support for dimensions," *Software Practice and Experience*, vol. 29, no. 7, pp. 605–616, 1999.

[31] G. Rosu and F. Chen, "Certifying measurement unit safety policy," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 2003, pp. 304–309.

[32] G. Biggs and B. A. MacDonald, "A pragmatic approach to dimensional analysis for mobile robotic programming," *Autonomous Robots*, vol. 25, no. 4, pp. 405–419, 2008.