PhysFrame: Type Checking Physical Frames of Reference for Robotic Systems[*]

Sayali Kate
Purdue University
USA
skate@purdue.edu

Michael Chinn
University of Virginia
USA
mec2wr@virginia.edu

Hongjun Choi
Purdue University
USA
choi293@purdue.edu

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

Sebastian Elbaum
University of Virginia
USA
selbaum@virginia.edu

ABSTRACT

A robotic system continuously measures its own motions and the external world during operation. Such measurements are with respect to some frame of reference, i.e., a coordinate system. A nontrivial robotic system has a large number of different frames and data have to be translated back-and-forth from a frame to another. The onus is on the developers to get such translation right. However, this is very challenging and error-prone, evidenced by the large number of questions and issues related to frame uses on developers' forum. Since any state variable can be associated with some frame, reference frames can be naturally modeled as variable types. We hence develop a novel type system that can automatically infer variables' frame types and in turn detect any type inconsistencies and violations of frame conventions. The evaluation on a set of 180 publicly available ROS projects shows that our system can detect 190 inconsistencies with 154 true positives. We reported 52 to developers and received 18 responses so far, with 15 fixed/acknowledged. Our technique also finds 45 violations of common practices.

CCS CONCEPTS

• **Software and its engineering** → **Abstract data types**; **Software defect analysis**.

KEYWORDS

Physical Frame of Reference, Frame Consistency, Type Checking, Static Analysis, z-score Mining, Robotic Systems

ACM Reference Format:
Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian Elbaum. 2021. PhysFrame: Type Checking Physical Frames of Reference for Robotic Systems. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468608

[*]An extended version is available at arXiv [19].

1 INTRODUCTION

Robotic systems have rapidly growing applications in our daily life, enabled by the advances in many areas such as AI. Engineering such systems becomes increasingly important. Due to the unique characteristics of such systems, e.g., the need of modeling the physical world and satisfying the real time and resource constraints, robotic system engineering poses new challenges to developers. One of the prominent challenges is to properly use physical frames of reference. Specifically, the operation of a robotic system involves moving individual body parts and interacting with the external world. It entails precisely measuring positions and orientations of body parts and external objects. All these measurements are represented with respect to a set of coordinate systems (also called *frames of reference* or *frames* in short). For example, in a three-dimensional coordinate system, the location (1,2,3) is meaningless unless we know the frame to which this location refers to. The origin of a frame provides the reference position (0,0,0), whereas the orientation of the three axes tells us the directions in which they point to. Further, the origin and orientation of a frame itself may be defined relative to another frame and so on.

Robotic systems usually follow a modular design, in which body parts and control software components are developed independently by various parties. As such, different components often use different frames. For example, a camera has the *camera frame* through which the physical world is measured from the camera's perspective. In this frame, the center of camera is the origin and the axes follow the orientations of the camera. The body of a robot has the *body frame* whose origin is the center of the body and axes are pre-defined based on the shape of robot. Measurements in the camera frame have to be translated to the body frame before they can be used in body related computation. Since the camera can be mounted at different places and moves during operation, such translation has to be done by the developers in software. More discussion about frames and an example can be found in Section 2.

Most robotic systems are developed in general purpose languages such as C/C++, facilitated by domain specific libraries. Such languages do not have intrinsic support for the additional complexity induced by the use of frames. For example, although popular libraries such as ROS [32] provide functions to facilitate translation between frames, the onus is on the developers to determine the reference frames of program variables, the places where translation

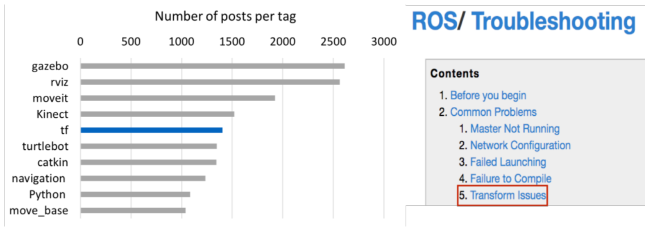


Figure 1: Forum posts that show frames in robotic programming are difficult to handle. (Note: right-side snapshot is taken from http://wiki.ros.org/ROS/Troubleshooting page licensed under the CC BY 3.0)

is needed, and correctly implement the concrete translations. Since translation is often done by vector/matrix operations, many developers even realize their own translation functions from scratch without using ROS APIs. As a result, use of frames is error-prone, even for experienced developers. This is evidenced by the fact that a large number of questions and issues for a robotic system project are usually regarding reference frames. Figure 1 shows that questions tagged with '*tf*' (a keyword for ROS frame support [14, 36]) are among the top ten types of questions by ROS developers, such as"*confused about coordinate frames*", "tf *transforms are confusing*", and "*is there an easier way to find the coordinates of a point in another frame*". Also, ROS wiki identifies transform issues as one of the top 5 common problems. Besides the difficulties of getting them right during development, misuse of frames may cause robot runtime malfunction [5], code reuse problems [1, 4], and maintenance difficulties after deployment.

ROS provides a number of tools to help developers debug frame related problems [2]. These are mostly runtime tools that can facilitate visualization of frames, relation between frames, details of a transformation between two frames (e.g. when it is created and by which component). However, it is always more desirable to detect problems as early as possible in the development life-cycle. A static tool that can scan and detect frame misuses before running the system would be highly desirable. To the best of our knowledge, there are unfortunately no such tools.

In this paper, we introduce and implement a novel fully automated static tool to identify frame related problems in ROS-based projects. Developers direct the tool to operate on a project directory, and the tool automatically models the project, and checks for potential frame faults. Since each physical state related variable (e.g., sensor reading, acceleration, velocity, and position) is associated with some frame, our tool builds on such such associations which are similar to variable types by nature. We hence propose a type system to model reference frames. A traditional type system often starts with type annotations that require developers to know variable types in the first place, which is difficult in our context as knowing the right frames is usually difficult. Instead, our technique automatically infers frames and performs type consistency checks, leveraging ROS conventions that can be extracted from its specifications and mined from code repositories. The technique uses frame information sources such as static data values exchanged between components and the ROS frame APIs for assigning frame types, and defines type propagation and checking rules based on the standard coordinate conventions documented in ROS-Enhancement-Proposals (REPs) and the semantics of frame related operations such as displacements and rotations. As part of our contribution we present a type language as a vehicle to describe these type rules , which constitute the foundation for our domain-specific static analyzer.

Our contributions are summarized as follows:

- We propose a fully automated type inference and checking technique for physical frames in ROS-based programs to detect frame inconsistencies and convention violations.
- We define frame and transformation abstract types, and our system automatically infers such types for variables. This is achieved by novel abstraction of frame related program semantics.
- We present a data-driven approach to identifying commonly followed practices that may not be documented anywhere. Our system also checks for violations of such practices.
- We implement a tool, PhysFrame, and evaluate it on 180 ROS projects from Github. It reports 190 type inconsistencies with 154 true positives (81.05%). We report 52 from projects that are recently active to developers and have received 18 responses so far, with 15 fixed/confirmed. It also detects 45 violations of common practices.

2 REFERENCE FRAMES AND ROS

2.1 Frames

A coordinate system that is used as a reference for measurements of quantities such as position and velocity is called a *frame of reference*, in short, a frame. It is defined by two components: an origin point and the axes system. A robotic system often consists of many parts, each having at least a *body frame* with the origin at the center of the rigid shape of the part, and axes pointing to some orthogonal directions. It allows measurements in the perspective of the part. The body frame for the base of a robot is often called *base_link*. It is one of the most important frames in a robot as measurements in this frame describe how the robot as a whole sees the world. The body frame for a camera is called the *camera frame*. If the part is a sensor, it often has other frames to denote sensor readings. For example, a camera has multiple *optical frames*, including the *color frame* measuring RGB format image, the *depth frame* measuring depth image, and the *ir frame* measuring infra red image. These frames are centered at the corresponding sensors (e.g., lens) instead of the camera body. They are different from the camera frame.

There are also the *odometry (odom) frame*, the *map frame*, and the *earth frame* to denote the physical world, i.e., how the world sees the robot and its surroundings. The *odom* frame represents robot's initial pose and does not move with the robot. It is like a third person standing at the robot's starting position and observing the motion of robot in the environment. The *map* frame projects everything in a localized map such as inside a room. It is like a third person standing somewhere in the room (not the starting point of the robot) and observing the robot and its environment. The *earth* frame is an earth centered frame.

During operation, sensor readings are acquired regarding the corresponding sensor frames. They are then transformed to the body frames of the parts where the sensors are installed, and eventually to the *base_link* frame and some world frame(s) in which control algorithms use the transformed values to compute actuation signals. These signals may undergo the inverse transformations until they reach the actuators. A simple commodity robot may have

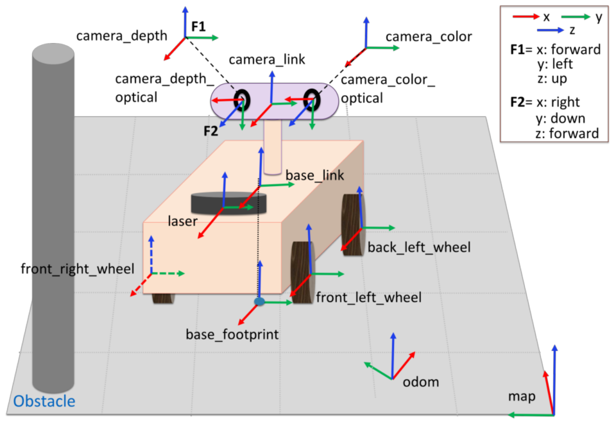


Figure 2: Frames in a Simple Robot

up to 81 different frames [35] as shown in Figure 7 in Appendix in [19] and values have to be correctly transformed between any pair of frames. It is a heavy burden for developers to get all these transformations right.

Figure 2 shows a simple robot. Observe that each of its parts has its own body frame (e.g., *camera_link*, *laser*, and *base_link*). There are also the world frames such as *odom* and *map*. Suppose if the vehicle locates an obstacle, then it knows the obstacle's position in the *camera_depth_optical* frame. This position is different from the position of the obstacle in the *map* frame. In order to answer the question "what is the position of an obstacle in the room?", the vehicle needs to know how the *camera_depth_optical* frame is positioned and oriented relative to the *base_link* frame and how the *base_link* frame is positioned and oriented relative to the *map* frame, then has to convert the obstacle position from the *camera_-depth_optical* frame all the way to the *map* frame. Moreover, the position of the obstacle in the frame of front right wheel, *front_-right_wheel*, is different from that in the *base_link* frame. Hence, to know the position with respect to the front right wheel, the vehicle needs to know the position and orientation of *front_right_-wheel* frame relative to the *base_link* frame. Different body frames may have different axis orientation conventions. Figure 2 presents two axis orientations labeled as F1 and F2. Each part has a default forward-facing direction (e.g., the nose of an aerial robot and the front of a camera), the values *forward, backward, left, right*, etc., are defined regarding the default direction. F1 and F2 are widely used in ROS. Moreover, various libraries use different conventions. For example, (x: right, y: up, z: backward) axis orientation is widely used in OpenGL.

2.2 ROS and Frames

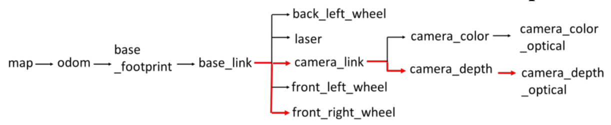
The Robotic Operating System (ROS) [32] is an open source software framework for building robotic applications. Its modular, distributed nature has made it widely used. Its users range from a novice developer working on a hobby project to industries. ROS provides various components such as drivers, localization packages, odometer packages developed by the community in addition to the core components. It provides a package called *tf* or *tf2* [14, 36] to support transformation between frames.

ROS Transforms. To specify frames and transformations, developers leverage the *tf* library to explicitly publish transformations between frames (called *transforms* in ROS), which also implicitly introduce the frames. A transform consists of a parent frame and a child frame, uniquely identified by their ids. ROS developers often say a transform is from the parent to the child. For example "*map*⟶*odom*" means a transform from the *map* frame to the *odom* frame. It is defined by providing the displacements of the child frame's origin and the rotation of its axes regarding the parent frame. However, *the transform is used to translate states in the child frame to the parent frame* (not the other way suggested by its name). We unfortunately have to inherit such term ambiguity from ROS. In the rest of the paper, we explicitly distinguish the name of a transform and its use in state translation to avoid confusion.

After transforms are published, other ROS components, such as standard control algorithms, subscribe to these transforms based on their names and use them in computation without knowing how they are implemented. Hence, these core algorithms can be agnostic to concrete system composition.

Launch Files. Many parts have fixed and static relative positions, e.g., camera mounted at a fixed position of an aerial robot. The transforms between their frames can be statically defined in *launch files*, which are a kind of static initialization files that define how to start executing a robotic system. The transforms for frames with dynamic relative positions, e.g., when a camera is mounted on a gimbal, have to be published and updated on-the-fly during operation. They are hence implemented by code. An example launch file snippet can be found in Appendix A in [19].

TF Tree. Transforms are potentially needed in between each pair of frames. However, specifying/implementing transforms for each pair entails substantial and error-prone efforts. ROS has a clean hierarchical design to reduce the complexity. If we consider a frame as a node and a transform between two frames as an edge, ROS requires all published frames and transforms to form a tree, that is, no cycles and each node having only one parent. We call it a *TF tree*. As such, the developer only needs to focus on realizing the transforms denoting individual edges. ROS then automatically constructs the transform from an arbitrary frame f_1 to another arbitrary frame f_2, by first performing the transforms from f_1 to the lowest common ancestor frame of the two, and then the transform from the ancestor to f_2. It can be easily inferred that such a design substantially reduces the developers' burden. The below figure shows the TF tree for the robot in Figure 2. Observe that while the developers only need to develop and publish 12 transforms (denoted by the edges), ROS allows 156 transforms from a frame to any other frame. The red edges form a transform path between *front_right_wheel* to *camera_depth_optical*, when the right wheel on the front of the robot uses data from the camera depth sensor.



2.3 Frame Conventions in ROS

All ROS projects follow certain conventions when manipulating frames. Violation to these conventions would cause integration, maintenance, and reuse problems. Our technique leverages these conventions to automatically infer and check frame types. There are two sources for extracting conventions: ROS specification and launch files. The former is explicit and the latter is implicit. Since ROS conventions are often related to static coefficients values (e.g., orthogonal axes orientations), they are difficult to extract directly from code where coefficients are largely variables.

Explicit Conventions From ROS Specifications. These conventions can be classified into the following three categories.

Naming Conventions. ROS applications use messages to exchange data between components. The `frame_id` field in a message specifies the frame for the data in that message. These ids follow certain conventions. For example, "`map`" is used for the map frame [27], and "`base_link`" for the body frame of the robot's base [27, 28]; optical frames should have an "`_optical`" suffix [15].

Axis Orientation Conventions. According to [15], body frames should have the (x: forward, y: left, z: up) or FLU axis orientation (i.e., F1 orientation in Figure 2); optical frames should have the (x: right, y: down, z: forward) orientation (i.e., F2 in Figure 2).

Tree Order Conventions. Recall that in a robotic system, the published frames and transforms are organized in a TF tree. ROS requires TF trees follow a partial order: *earth* ⟶ *map* ⟶ *odom* ⟶ *base_link* [27]. Note that while a concrete TF tree for a specific robotic system may omit some of these frames and have additional frames, such an order must be respected. The reason is that the order optimizes the performance for mostly commonly seen transforms.

Implicit Conventions from ROS Launch Files. Launch files provide a rich resource for mining implicit conventions. We focus on extracting two kinds of implicit conventions.

Co-occurrence Conventions. Robotic systems of a same kind share similarity in their physical compositions. As such, they often use similar frames and transforms. This is reflected by co-occurrences of transforms in launch files. For example, a ground vehicle robot with two wheels has one wheel on its left and the other wheel on its right. If a transform from *base_link* to the frame of left wheel is defined, then a transform from *base_link* to the frame of right wheel is also defined.

Value Conventions. Certain transforms are associated with fixed coefficient values such as 0 rotation angles or 0 displacement values. For example, a transform from *camera_depth* to *camera_depth_optical* must have 0 displacement values.

3 MOTIVATION

To motivate our technique we use code from the *simultaneous localization and mapping* (SLAM) component ORB2-SLAM2 (links in the caption of Figure 3) [3, 29]. Localization and mapping algorithms like ORB are key to the operation of modern mobile robots, incrementally building a map of the robot surroundings utilizing its *camera* as the robot navigates. This component, however, uses different conventions from ROS and hence the most important task for interfacing then is to perform appropriate frame conversions.

The code snippet in Figure 3 aims to implement a transform to facilitate translation of states from the ROS *camera* frame to the ROS *map* frame using the environment model of ORB-SLAM2. The conversion implementation, however, is incorrect, and PhysFrame catches this error by type checking.

Constructing the aforementioned transform is non-trivial, and consists of three groups of operations. First, given some ROS state in the child *camera* frame with the conventional FLU orientation, it should be translated into ORB's RDF orientation, (x:right, y:down, z:forward). Second, the position of the *camera* frame in the *map* frame needs to be determined. However, since ORB-SLAM2 models the environment in the *camera* frame, one can only query the position of *map* frame in the *camera* frame, so it needs to be transposed to acquire the position of *camera* frame in the *map* frame. Third, the transform should translate the results back to ROS's FLU orientation. The code shown fails to accomplish the last step.

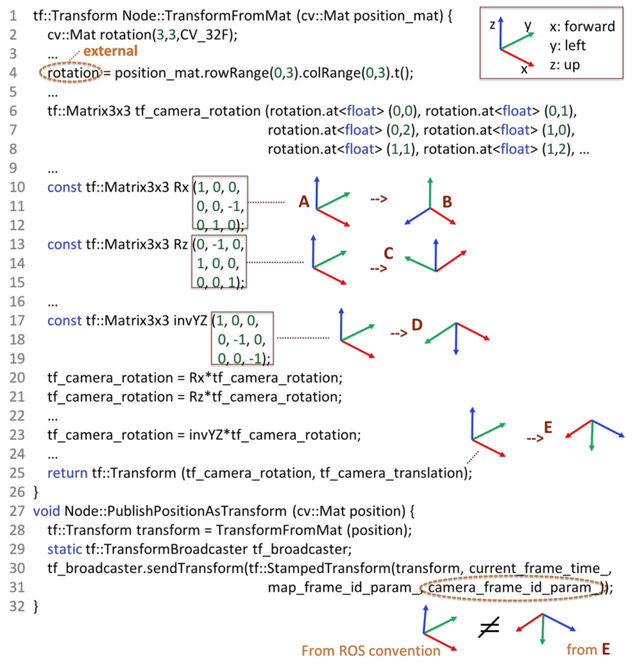

```
1  tf::Transform Node::TransformFromMat (cv::Mat position_mat) {
2    cv::Mat rotation(3,3,CV_32F);
3    ...
4    rotation = position_mat.rowRange(0,3).colRange(0,3).t();
5    ...
6    tf::Matrix3x3 tf_camera_rotation (rotation.at<float> (0,0), rotation.at<float> (0,1),
7                                      rotation.at<float> (0,2), rotation.at<float> (1,0),
8                                      rotation.at<float> (1,1), rotation.at<float> (1,2), ...
9    ...
10   const tf::Matrix3x3 Rx (1, 0, 0,
11                           0, 0, -1,
12                           0, 1, 0);
13   const tf::Matrix3x3 Rz (0, -1, 0,
14                           1, 0, 0,
15                           0, 0, 1);
16   ...
17   const tf::Matrix3x3 invYZ (1, 0, 0,
18                              0, -1, 0,
19                              0, 0, -1);
20   tf_camera_rotation = Rx*tf_camera_rotation;
21   tf_camera_rotation = Rz*tf_camera_rotation;
22   ...
23   tf_camera_rotation = invYZ*tf_camera_rotation;
24   ...
25   return tf::Transform (tf_camera_rotation, tf_camera_translation);
26 }
27 void Node::PublishPositionAsTransform (cv::Mat position) {
28   tf::Transform transform = TransformFromMat (position);
29   static tf::TransformBroadcaster tf_broadcaster;
30   tf_broadcaster.sendTransform(tf::StampedTransform(transform, current_frame_time_,
31                                map_frame_id_param_, camera_frame_id_param_));
32 }
```

Figure 3: Example of incorrect transform between frames
source: https://git.io/JtSHb, fixed source: https://git.io/JtSRt

Function `PublishPositionAsTransform()` starting at line 27 takes the ORB-SLAM2's position matrix (denoting the *map* frame's position in the *camera* frame), i.e., variable `position`, and constructs a ROS transform (line 28) by invoking `TransformFromMat()`, which starts at line 1. In line 4 it acquires the rotation matrix from the ORB-SLAM2's position matrix, which includes both displacements and rotation, and performs transposition by function `t()`. Line 6 clones it to `tf_camera_rotation`. Line 10 creates a matrix denoting rotation of an FLU axis system anti clockwise for 90 degree with respect to the *x* axis (shown to the right of the code). Lines 13 and 17 create two additional rotation matrices. Lines 20-23 compose the three matrices and the earlier rotation matrix. Line 25 creates a ROS transform from displacements denoted by `tf_camera_translation` and rotation denoted by `tf_camera_rotation`. When the transform is used in state translation, a state is multiplied with `tf_camera_rotation`, which equals to `invYZ*Rz*Rx*rotation`. The first three matrixes are equivalent to transforming the FLU orientation of the state to the RDF orientation as shown on the right of line 25. The fourth matrix (`rotation`) then translates the state from the camera frame to the map frame. However, the constructed transform is problematic as it forgets to further translate back to FLU. This bug will cause downstream system mis-behaviors, which could be devastating if the camera

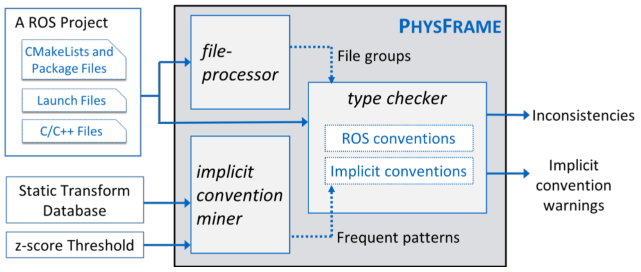


Figure 4: Overview of PhysFrame

readings are the dominant source for control decisions. It is fixed by appending the inverse rotation matrices of `invYZ`, `Rz`, and `Rx` to `tf_camera_rotation` after line 23, to change RDF to FLU.

PhysFrame associates each state variable with a frame type that abstracts both the displacements and orientation of the frame in its parent frame. It also associates each transform variable with a transform type that denotes the displacements and rotation needed in frame translation. It propagates such types following operation semantics and checks consistency. In the example code, it associates matrix `Rx` at line 10 with a transform type from the FLU orientation A to the orientation B. The type is derived from the constant matrix values. It further associates matrix `Rz` at line 13 with a transform type from FLU to the orientation C; and `invYZ` at line 17 from FLU to D. By modeling the semantics of matrix multiplications at lines 20, 21, and 23, it determines that the transform at line 25 has the type from FLU to RDF. This yields a type error at lines 30-31 when the transform is published by function `sendTransform()` because ROS convention demands both camera frame and map frame to have FLU, whereas the transform converts FLU to RDF.

Observe that it is difficult to get such transformation right due to its intrinsic subtlety and complexity. While this is only a one-step transform, as mentioned earlier, a non-trivial robotic system has more than 80 different frames and transforms between any pair may be necessary. The example bug is just one of the many kinds of bugs PhysFrame detects. Others include missing frame and broken TF tree. More discussion can be found in Section 4.2.3.

4 OUR APPROACH

4.1 Overview

Figure 4 presents an overview of PhysFrame. It takes as input a C++ ROS project, and outputs frame inconsistencies and implicit convention violations. PhysFrame consists of three components: *implicit convention miner*, *file-processor*, and *type checker*.

The miner takes a database of static transforms extracted from the launch files of a large repository of over 2200 mature ROS projects from Github and performs data mining to infer implicit conventions. The miner identifies frequent static patterns that include pair of transforms that commonly co-exist in a project, transforms commonly having zero displacements, and transforms commonly having zero rotation. Since these patterns may happen coincidentally, the miner computes a frequency for each pattern to quantify its certainty (we use z-score [13] as a frequency measure) and reports it if it is above a specified z-score threshold. The mining process is executed once on the collected static transform database, but can be updated as new static transforms are incorporated into the database. The produced rule conventions are then used in the analysis for each subject ROS project to identify anti-patterns. Details can be found in Appendix B in [19] .

A ROS project often contains multiple executable subsystems, each having its own TF-tree. PhysFrame has to analyze these subsystems one at a time. The file-processor separates a project into subsystems, each consisting of a group of C/C++ source files and launch files that are executed together. A subsystem usually has a separate initial launch file. The file-processor hence starts from these top level launch files, identifies the child launch files and the corresponding CMakeLists files, which indicate the source files involved in each subsystem, and in turn finds file groups that represent the project's subsystems. Details are elided.

The type checker then takes patterns and file groups information, together with project files, and types variables and statements in the C/C++ source files. Type errors are reported when the program cannot be properly typed. PhysFrame reports 7 kinds of type errors and 2 kinds of type warnings as inconsistencies, and reports 3 kinds of implicit convention violation warnings (see Section 4.2.3). Warnings may not lead to broken functionalities but problems in code maintenance and reuse. In the following, we will focus on explaining the type system.

4.2 Type System

Before introducing the formal language and type rules, we intuitively explain how frames and transforms are used. Typically, one-step transform objects are explicitly created to facilitate translation of states in a child frame (e.g., position) to a parent frame. A transform specifies the relative position and orientations of the child frame in the parent frame, denoted by displacements and rotations. It also specifies the (string) ids for the child and parent frames. An id is the unique representation of a frame. The creations of transforms also implicitly define frames (through the ids). They are implicit as there are no explicit ROS data structures for frames. A state variable can be explicitly associated with some frame id, yielding a *stamped* (a ROS term) state variable. A transform can be explicitly applied to some stamped variable (in a child frame) to acquire its correspondence in the parent frame. One-step transforms can be published and hence become part of the global TF tree. As such, transforms between arbitrary frames (e.g., those multiple steps apart) can be automatically constructed by traversing the tree (Section 2.2). To define a one-step transform, developers often explicitly specify the entailed rotation. A rotation can be composed from others. For example, a 60 degree rotation can be composed from two 30 degree rotations. Plain state variables (not stamped) can be operated on just like regular C/C++ variables/data-structures. As such, even they (implicitly) belong to some frames, frame inconsistencies cannot be detected. Furthermore, ROS does not provide type checking mechanism even for stamped variables. The C/C++ type system cannot offer help either as it does not have any domain knowledge. For example, it can type a variable to a ROS transform data structure but cannot identify the specific transform type, i.e., the parent and child frames and the displacements and rotation involved.

4.2.1 A Simplified Language and Types. ROS is based on C/C++ with a set of library functions. While our type system supports the complex syntax of C/C++/ROS and models the frame related APIs, we use a simplified language for discussion brevity. The language focuses on modeling frame related semantics and ignores the standard C/C++ types, operations and statements. It directly models a number of ROS *tf* APIs as language primitives. Moreover, since our analysis is flow insensitive, control structure like conditional statements and loops are less relevant and hence not modeled.

Types. We define three types: (1) frame type (*Frame*) that represents the frame information of a state variable, (2) transform type (*Transform*) that represents frame transforms, and (3) rotation type (*Rotation*) that represents changes in the axes' directions. Note that these are not ROS types, but rather types in PhysFrame.

Frame Type. This is an abstract type and always used with some concrete state type such as *Point* and *Pose*, indicating points and poses in a specific frame, respectively. For example, ROS uses `tf::Stamped<T>(T x, ..., String id)` to denote a `T` type state object associated with a frame identified by `id`. The frame type's definition is as follows.

$$Frame ::= < id, pid, x, y, z, d_x, d_y, d_z > \qquad (1)$$

It is composed of eight fields: *id* denoting the (string) name of frame, *pid* the name of its parent frame type; the next three fields, x, y, z, being the positions of the frame's origin with respect to its parent frame, where their values are of type *Displacement*.

$$Displacement ::= R \mid \top \qquad (2)$$

where, R is a static numerical value, and $\top$ represents a value unknown statically. The next three fields, d_x, d_y, d_z, encode the x, y, z axes orientations of a frame, with values of the *Orientation* type:

$$Orientation = \{\textit{left, right, up, down, forward, backward}\}$$

They denote the (orthogonal) orientations of axes *regarding the body part corresponding to the frame.* They are independent of the placement of the body part regarding its base. For example, although a camera may be installed with arbitrary (and even dynamic) angles regarding a robot's body, the orientation of the camera frame's axes is *forward-left-up* (FLU), which is orthogonal regarding the camera. Although axes orientations are orthogonal, different components and robotic frameworks have different orientation conventions, transformations need to be explicitly performed by developers.

Transform Type. ROS allows developers to create transform objects that denote transformation from a frame to another (e.g., using the `tt::Transformer` class). We associate such objects with a transform type that abstracts information to facilitate consistency and convention checks. The transform type is defined as follows.

$$Transform ::= < cid, pid, x, y, z, \tau_r > \qquad (3)$$

Here, *cid* and *pid* are the child and parent frame ids, respectively, x, y, z are abstract values of the aforementioned *Displacement* type, and τ_r denotes an abstract rotation, which will be explained next. The displacements and rotation together define a transformation.

Rotation Type. Rotations change axes orientations of a frame. ROS allows creation of first-order rotation objects (e.g., through the ROS `tt::Transform` class) and manipulations of these objects. They can be further used to construct transform objects. In this work, we are interested in orthogonal rotations such as those with 90 or 180 degrees as they are used in changing axes orientations. We hence define rotation type as follows.

$$Rotation ::= < ds_x, ds_y, ds_z > \ \mid \top \qquad (4)$$

The three fields, ds_x, ds_y, ds_z are variables of *DirSwitch* type:

$$DirSwitch = \{``x", ``-x", ``y", ``-y", ``z", ``-z", \top\}$$

Field ds_x holds a value indicating that the axis denoted by the value points in the same direction as the previous x axis (i.e., the x axis before rotation). Fields ds_y and ds_z work in a similar way. We use the following example to explain the semantics. Let $< ``z", ``-x", ``-y" >$ be a *Rotation* type. It switches orientations of axes such that z axis points in the same direction as the previous x axis' direction, $-x$ axis points in the same direction as the previous y axis' direction (i.e., the current x axis is the opposite of the previous y axis), and $-y$ points in the same direction as the previous z axis' direction. Assume a variable of frame type < *..., forward, left, up*>. After applying the aforementioned rotation, a new frame type < *..., right, down, forward* > is derived. Value $\top$ means the rotation is not orthogonal or cannot be determined statically.

$$\begin{array}{lll}
\langle Program \rangle & P & ::= S \\
\langle Statement \rangle & S & ::= S_1 ; S_2 \mid x := v \mid x := y \mid x := y \ \mathbf{op} \ z \mid \\
 & & fx := \mathbf{stamped}(s, x) \mid fx.id := s \mid \\
 & & x := \mathbf{get\_data}\ (fx) \mid fx := fy \mid fx := \mathbf{external}() \mid \\
 & & fx := \mathbf{transform\_to}(s, fy) \mid \\
 & & tx := \mathbf{new\_transform}(s_{chd}, s_{pnt}, x, y, z, rx) \mid \\
 & & \mathbf{sendTransform}\ (tx) \mid \\
 & & tx := \mathbf{lookupTransform}\ (s_{pnt}, s_{chd}) \mid \\
 & & fx := \mathbf{apply\_transform}\ (tx, fy) \mid rx := v_{3\times 3} \mid \\
 & & rx := x \mid rx := ry \otimes rz \mid \mathbf{publish}\ (s_{topic}, fx) \\
<StateVar> & x, y, z & \\
<StampedStateVar> & fx, fy, fz & \\
<TransformVar> & tx, ty, tz & \\
<RotationVar> & rx, ry, rz & \\
<ConstVector> & v & \\
<ConstString> & s &
\end{array}$$

Figure 5: Language

Language. Figure 5 presents the language. We call all ROS state variables *StateVar*, such as `tf::Point` and `tf::Pose`. State variables are frame agnostic. When they are explicitly associated with some frame, they become *StampedStateVar*[1], corresponding to `tf::Stamped⟨Point⟩` and `tf::Stamped⟨Pose⟩`, etc. in ROS. We use x, y, z for *StateVar* and *fx, fy, fz* for *StampedStateVar*. We also use variables *tx*, *ty*, *tz* and *rx*, *ry*, *rz* to denote transform and rotation variables. Note that these variables are not typed. Their types will be resolved by our type rules. We distinguish different kinds of variables just for better readability.

Statement "$x := v$" denotes a state variable assignment using a constant vector. A scalar variable can be considered as a vector variable with one dimension. Function **stamped()** explicitly associates x with a frame denoted by a string s, corresponding to `tf::Stamped⟨T⟩::Stamped()` in ROS. ROS allows stamping a state variable with an empty id and later explicitly setting the id field of the stamped variable. A common error is that the developer stamps an empty frame and later forgets to set the frame. Statement "*fx.id* := s" models the set frame operation. Function **get_data()** retrieves the plain state from a stamped state variable. Function **external()** models the situations where stamped state variables are

[1]ROS uses the term "*stamped*" to denote that a variable is contextualized with a frame. We hence use a similar term here.

published by libraries without source code and hence beyond our analysis. Function **transform_to()** transforms *fy* to a target frame denoted by *s*, yielding a new stamped variable *fx*. It corresponds to ROS functions such as `tf::Transformer::transformPoint()`. Function **new_transform()** creates a one-step transform object for translating states in the child frame to their correspondences in the parent frame. It specifies the parent and child frame ids (by s_{pnt} and s_{chd}, respectively), the displacements (by state variables x, y, z), and the rotation (by rx). The displacements and rotation specify the linear and angular positions of the child frame with respect to the parent frame. This primitive corresponds to creating a ROS `tf::StampedTransform` object. One-step transforms can be published by function **sendTranform()** such that they become edges of the TF tree (Section 2.2). Function **lookupTransform()** finds a transform from frame s_{pnt} to frame s_{chd}. These two frames may not correspond to any published one-step geometric transforms. Recall that ROS automatically constructs a transform by finding a path from s_{pnt} to the lowest common ancestor (in the TF tree) and then to s_{chd}. It corresponds to `tf::Transfomer::lookupTransform()`. Function **apply_transform()** applies a transform *tx* to a stamped variable *fy*. As such, *fy* must have the child frame type of *tx* and *fx* have the parent frame type of *tx*. Statement "$rx := v_{3\times3}$" specifies a constant rotation by a 3 times 3 matrix. One can also specify a dynamic rotation from a (matrix/vector) variable. Two rotations can be aggregated to one by matrix multiplication ⊗. A stamped variable can be published to a topic. Any (remote) subscribes of the topic will receive the value. As such, the variable's frame must be properly set. Otherwise, remote parties cannot make sense of it.

While PhysFrame supports conditionals, loops, functions, and other frame related ROS APIs. They are elided from our language.

4.2.2 Type Rules. Table 1 presents the type rules. These rules infer types for state variables and frame related variables and check for any inconsistencies, that is, variables cannot be properly typed, e.g., a variable having more than one types following the rules. We use meta-variables τ, τ_t, and τ_r to range over the infinite sets of *Frame*, *Transform*, and *Rotation* types. In our rules, a variable $x : \tau$ means that x has τ type. A statement $x := ... : \tau$ means that the statement has τ type, which is equivalent to the left-hand-side variable x having τ type as well. In general, a rule is read as follows: if the premises are satisfied (e.g., type checked), the conclusions are yielded. The rules are driven by the syntax of the language.

Data Assignment Rules. Rule R_1 specifies that for a copy statement, if the right-hand-side is typed to τ, the left-hand-side is typed to τ as well. Rule R_2 requires that in a binary operation of state variables, the two operands must have the same type τ. Then we conclude the result of the statement has the same type. Rule R_3 specifies that if x is typed to τ, which has the s as the frame id, we can conclude *fx* has the same type τ. Note that type inference is flow-insensitive, if x was free (i.e., untyped), the rule types it to τ. Rule R_4 types an explicit set frame statement to the type denoted by the id string s. Rule R_5 specifies that if a state variable x is acquired from a stamped variable *fx*, x inherits the type of *fx*. Rule R_6 specifies two stamped variables in a copy statement must have the same type.

Transformation-related Rules. Rule R_7 types a transformation statement. Specifically, the meaning of function *reachable*() is defined in Figure 6. It asserts that there is a direct/indirect transform from *fy*'s frame τ_1 to the frame τ denoted by s. If so, we conclude that *fx* has τ type. Rule R_8 types a statement that creates a one-step transform from $< s_{chd}, s_{pnt}, x, y, z, rx >$. It entails the introduction of an implicit frame type τ_c whose id is s_{chd}, parent id is s_{pnt}, displacements are those abstracted from x, y, z using the *disp*() function defined in Figure 6, and axes orientations are resulted from applying the (abstract) rotation specified by the rotation type τ_r of rx to the orientations of the parent frame. Function *rotate* is defined in Figure 6. It takes a 3-dimension source orientation (with values *left*, *right*, *forward*, etc.) and a 3-dimension rotation (with values "*x*", "*-y*", etc.), and produces a resulted 3-dimension target orientation. Specifically, the result x orientation, denoted as n_x is determined by searching for which rotation dimension has value of "*x*" or "*-x*". If the y rotation has value "*x*", denoted as ds_y = "*x*", n_x has the same orientation as the original y orientation (i.e., d_y). If the value is "*-x*", the orientation is flipped. An example of such rotation can be found in the discussion of rotation type at the end of Section 4.2.1.

Rule R_9 types a statement that publishes a transform *tx*. It requires the edge from the parent frame to the child frame does not break the TF tree, e.g., by introducing cycles or having multiple parents for a child. The *tree_form_with_edge*() function not only checks the condition but also adds the edge to the TF tree if the condition is satisfied. Rule R_{10} types a statement that looks up a transform from an arbitrary parent frame to an arbitrary child frame in the TF tree. The transform may not be any of the created one-step transforms, but rather automatically composed by ROS by traversing the tree. It requires reachability between the two frames (in the tree), which implicitly demands the existence of the two

reachable(s_1, s_2): s_1 and s_2 share some common ancestor in the TF tree and hence reachable from each other

$$disp(x) = \begin{cases} r & x \text{ holds a constant } r \in R \text{ after constant propagation} \\ \top & \text{otherwise} \end{cases}$$

$rotate(< d_x, d_y, d_z >, < ds_x, ds_y, ds_z >) = < n_x, n_y, n_z >$, with

$$n_x = \begin{cases} d_x & ds_x = \text{“}x\text{”} \\ opposite(d_x) & ds_x = \text{“}-x\text{”} \\ d_y & ds_y = \text{“}x\text{”} \\ opposite(d_y) & ds_y = \text{“}-x\text{”} \\ d_z & ds_z = \text{“}x\text{”} \\ opposite(d_z) & ds_z = \text{“}-x\text{”} \\ \top & otherwise \end{cases} \quad n_y = ..., \quad n_z = ...$$

tree_form_with_edge(s_1, s_2): adding an new edge denoted by $s_1 \longrightarrow s_2$ does not break the TF tree form

$$orthogonalize(v_{3\times3}) = \begin{cases} \top & ds_x = \top \vee ds_y = \top \vee ds_z = \top \\ \langle ds_x, ds_y, ds_z \rangle & otherwise \end{cases}$$

$$ds_x = \begin{cases} \text{“}x\text{”} & v_1 = [1, 0, 0] \\ \text{“}-x\text{”} & v_1 = [-1, 0, 0] \\ \text{“}y\text{”} & v_1 = [0, 1, 0] \\ \text{“}-y\text{”} & v_1 = [0, -1, 0] \\ \text{“}z\text{”} & v_1 = [0, 0, 1] \\ \text{“}-z\text{”} & v_1 = [0, 0, -1] \\ \top & otherwise \end{cases} \quad ds_y = ... \quad ds_z = ...$$

$$compose(\langle ds'_x, ds'_y, ds'_z \rangle, \langle ds''_x, ds''_y, ds''_z \rangle) = \langle ds_x, ds_y, ds_z \rangle$$

$$ds_x = \begin{cases} ds''_x & ds'_x = \text{“}x\text{”} \\ -ds''_x & ds'_x = \text{“}-x\text{”} \\ ds''_y & ds'_x = \text{“}y\text{”} \\ -ds''_y & ds'_x = \text{“}-y\text{”} \\ ds''_z & ds'_x = \text{“}z\text{”} \\ -ds''_z & ds'_x = \text{“}-z\text{”} \\ \top & otherwise \end{cases} \quad n_y = ..., \quad n_z = ...$$

Figure 6: Auxiliary functions used in type rules

Table 1: Type Rules

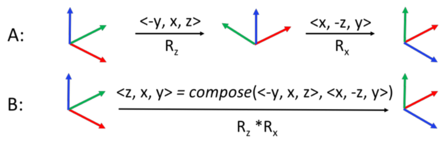
R_1	$\dfrac{y : \tau}{x := y : \tau}$	R_2	$\dfrac{y : \tau \quad z : \tau}{x := y \textbf{ or } z : \tau}$
R_3	$\dfrac{\tau.id = s \quad x : \tau}{fx := \textbf{stamped}(s, x) : \tau}$	R_4	$\dfrac{\tau.id = s}{fx.id := s : \tau}$
R_5	$\dfrac{fx : \tau}{x := \textbf{get\_data}(fx) : \tau}$	R_6	$\dfrac{fy : \tau}{fx := fy : \tau}$
R_7	$\dfrac{\tau.id = s \quad reachable(\tau.id, \tau_1.id) \quad fy : \tau_1}{fx := \textbf{transform\_to}(s, fy) : \tau}$		
R_8	$\dfrac{rx : \tau_r \;\; \tau_c = \langle s_{chd}, s_{pnt}, disp(x), disp(y), disp(z), d_x, d_y, d_z \rangle \quad s_{chd} != s_{pnt} \quad \tau_p.id = s_{pnt} \quad \langle d_x, d_y, d_z \rangle = rotate(\langle \tau_p.d_x, \tau_p.d_y, \tau_p.d_z \rangle, \tau_r)}{tx := \textbf{new\_transform}(s_{chd}, s_{pnt}, x, y, z, rx) : \; < s_{chd}, s_{pnt}, disp(x), disp(y), disp(z), \tau_r >}$		
R_9	$\dfrac{tx : \tau_t \quad tree\_form\_with\_edge(\tau_t.pid, \tau_t.cid)}{\textbf{sendTransform}(tx) : \tau_t}$		
R_{10}	$\dfrac{reachable(s_{pnt}, s_{chd}) \quad \tau_t.pid = s_{pnt} \quad \tau_t.cid = s_{chd}}{tx := \textbf{lookupTransform}(s_{pnt}, s_{chd}) : \tau_t}$		
R_{11}	$\dfrac{tx : \tau_t \quad fy : \tau \quad \tau_t.pid = \tau'.id \quad \tau_t.cid = \tau.id}{fx := \textbf{apply\_transform}(tx, fy) : \tau'}$		
R_{12}	$\dfrac{\tau_r = orthogonalize(v_{3\times3})}{rx := v_{3\times3} : \tau_r}$		
R_{13}	$\dfrac{}{rx := x : \top}$		
R_{14}	$\dfrac{\tau'_r = \langle ds'_x, ds'_y, ds'_z \rangle \quad \tau''_r = \langle ds''_x, ds''_y, ds''_z \rangle \quad ry : \tau'_r \quad rz : \tau''_r \quad \tau_r = compose(\tau'_r, \tau''_r)}{rx := ry \otimes rz : \tau_r}$		
R_{15}	$\dfrac{ry : \langle ds_x, ds_y, ds_z \rangle \quad rz : \top}{rx := ry \otimes rz : \langle ds_x, ds_y, ds_z \rangle}$		
R_{16}	$\dfrac{fx : \tau}{\textbf{publish}(s_{topic}, fx) : \tau}$		

frames. Rule R_{11} specifies that when typing a statement applying a transform *tx* to a stamped variable *fy*, the transform's child frame must equal to *fy*'s frame, and its parent frame is the resulting frame.

R_{16} types a statement that publishes a stamped variable. It requires that the variable is properly typed.

Rotation-related Rules. Rule R_{12} types the creation of a rotation from a 3×3 constant vector. Function *orthogonalize*() is used to abstract the vector to the corresponding *DirSwitch* abstract values. Its definition can be found in Figure 6. Specifically, an orthogonal 3×3 vector is composed of three orthogonal unit vectors, i.e., each row and each column has only one non-zero element of value 1 or -1. Now, if we consider that rows 1,2,3 represent initial x, y, z orientation and columns 1,2,3 represent then changed x, y, z orientation, then the column position of the non-zero element in a row indicates the axis of the changed orientation that is the same as the axis of the initial orientation represented by the row. For example, value -1 in row 2 and column 1 denotes that the direction of y axis of initial orientation becomes the direction of $-x$ axis of the changed orientation. When the matrix does not denote orthogonal rotation, the resulted type is $\top$.

Rule R_{13} types a rotation creation from a variable (i.e., non-constant). It usually corresponds to angular transformation that is needed for state translation across frames, but not for orientation changes. For example, if a camera can spin regarding the body, its readings (regarding the camera frame) have to go through a dynamic angular transformation in order to be interpreted regarding the body frame. Note that in this case, the angular transformation is independent of frame orientations. Both the camera and the body frames have orthogonal orientations. Rule R_{14} types a rotation composition statement. It leverages function *compose*() to derive a new rotation type. In the function definition in Figure 6, $\langle ds'_x, ds'_y, ds'_z \rangle$ and $\langle ds''_x, ds''_y, ds''_z \rangle$, both denote change in the orientation, where the former change is followed by the latter. For example, Rz and Rx vectors in our motivating example (Figure 3) represent $\langle -y, x, z \rangle$ and $\langle x, -z, y \rangle$ changes respectively as per rule R_{12}. Part A in the below figure shows these orientation changes in a sequence; whereas part B shows the orientation change that is computed by aggregating these two changes. Note that both parts result into the same orientation.



R_{15} types a rotation composition in which one of the rotations is not static or not orthogonal. Note that an arbitrary angular rotation can be integrated with an orthogonal axes orientation rotation. Since we only model and check axes orientation, the resulted type is the one that represents orthogonal orientation changes.

Table 2: Type checking of example in Figure 3

St#	Statement	Type	Rule
6	$tf\_cam\_rotation := tmp$	$\top$	R_{13}
10	$Rx := [[1, 0, 0], [0, 0, -1], [0, 1, 0]]$	$\langle x, -z, y \rangle$	R_{12}
13	$Rz := [[0, -1, 0], [1, 0, 0], [0, 0, 1]]$	$\langle -y, x, z \rangle$	R_{12}
20	$tf\_cam\_rotation := Rx \otimes tf\_cam\_rotation$	$\langle x, -z, y \rangle$	R_{15}
21	$tf\_cam\_rotation := Rz \otimes tf\_cam\_rotation$	$\langle z, x, y \rangle$	R_{14}
23	$tf\_cam\_rotation := invYZ \otimes ...$	$\langle z, -x, -y \rangle$	R_{14}
25	$rtmp := tf\_cam\_rotation$	$\langle z, -x, -y \rangle$	
30	$ttmp := \textbf{new\_transform}(s_{cam}, s_{map}, ..., rtmp)$	error because $\textbf{rotate}(\langle f, l, u \rangle, \langle z, -x, -y \rangle) = \langle r, d, f \rangle$	R_8

Example. Table 2 presents part of the example code in Figure 3 written in our language and the types computed by the type rules. The first column shows the original line numbers of the statements. St# 6 is an assignment of a rotation variable from a state variable. The state variable *tmp* gets a value from the external value state variable *rotation*. Therefore, as per rule R_{13}, *tf_cam_rotation* is typed to $\top$. St# 10 and 13 are the assignments of rotation variables from constant vectors. Rule R_{12} allows typing *Rx* and *Rz* to $\langle x, -z, y \rangle$ and $\langle -y, x, z \rangle$, respectively. Further, St# 20 and 21 denote the composition of rotations: the former is typed with the type of *Rx* by R_{15} as the type of *tf_cam_rotation* is $\top$, whereas the latter is typed with $\langle z, x, y \rangle$, which is computed by composing types of *Rz* and *tf_cam_rotation* using rule R_{14}[2]. St# 23 and 25 are similarly typed. Finally, St# 30 constructs a new transform object. However, it cannot be properly typed by rule R_8 as applying rotation $\langle z, -x, -y \rangle$ to the parent map frame with the conventional orientation ⟨*forward*, *left*, *up*⟩, i.e, FLU, yields the orientation ⟨*right*, *down*, *forward*⟩, contradicting with the ROS orientation convention FLU for the camera frame. On the other hand, PhysFrame correctly type-checks the fixed version.

Soundness and Completeness. The essence of PhysFrame is to leverage flow-insensitive static analysis (like type checking) to

[2]PhysFrame uses SSA to handle variable re-definitions.

find frame related bugs. It is automated and does not require any user type annotations for subject ROS projects. As such, the type system is neither sound nor complete. This is because some type rules are uncertain. For example in rule R_{12}, a matrix representing orthogonal angular transformation may not be intended for frame axes orientation changes, but rather because the parent and child body parts are installed in a way that they form some orthogonal angle(s), while their frame axes orientations are identical (both FLU). However, since there is no way for PhysFrame to know the real meanings of such rotation without user input, it considers this an axes orientation rotation, which is the most common case. And stamped variables from third party libraries are widely used and PhysFrame may not be able to type them. Therefore, PhysFrame has both false positives and false negatives. On the other hand, we foresee future robotic system development frameworks may support full-fledged frame types which the developer can use to declare their variables. With such explicit type declarations, a sound and complete type system can be developed.

4.2.3 Frame Related Problems. PhysFrame reports 7 kinds of type errors, 2 kinds of type warnings and 3 kinds of violations of implicit conventions. The first two are also called inconsistencies.

Type Errors. This first kind is *TF tree related errors.* These errors include *a frame having multiple parent frames*, *frames forming a cycle*, and *frames not following conventional orders* such as a map frame must be the ancestor of a body frame (Section 2.3).

The second kind of type errors is *incorrect frames or transforms.* ROS allows developers to stamp a state variable x with an empty id. If these variables are published, meaning that other components from different parties may receive such data, they should be stamped with a proper frame (rule R_{16}). Moreover, if they are used in transforms, runtime errors will be triggered as well. In these cases, PhysFrame reports a *missing frame error.* Although not explicitly modeled by our rules, a few ROS data types require explicitly specifying child frame ids. If child frames are not set, PhysFrame reports *missing child frame errors.*

A transform is required to have different parent and child frames (rule R_8). Otherwise, it is meaningless. PhysFrame reports a *redundant transform.* Rule R_8 requires that the displacements and rotation in a newly created transform should yield (relative) positions and axes orientations consistent with the parent frame and ROS conventions (e.g., body frames should have *forward-left-up* (FLU) orientations). Otherwise, it reports *incorrect transforms.*

Type Warnings. A sensor transform that defines a sensor frame with respect to some other robotic body part like base_link, wrist, etc. or vice versa should have at least some non-zero displacement(s)/rotation(s) as a sensor's center unlikely aligns with a body part's center. PhysFrame reports a warning when a sensor transform has zero displacements and rotations. For a transform from parent to child, ROS naming conventions follow templates '<parent>_to_<child>' and '<child>_in_<parent>', which give the clear meanings of the transform. Developers however often inverse the parent and the child in transform variable names. PhysFrame hence reports *reversed name warnings.* These two are warnings as they may not affect the functionalities but rather degrade maintainability. Such checkings are not explicitly modeled in type rules although PhysFrame supports them.

Implicit Convention Violations. As mentioned in Section 2.3, implicit conventions are mined from launch files. As such, they may not be required but rather commonly seen. PhysFrame reports three kinds of warnings. The first is *null displacement expected* warning, meaning that certain transforms or frames are supposed to have null displacements. The second is *null rotation expected* warning, meaning that certain transforms are supposed to have null rotations. The third is that *co-occurrence violation* warnings when frames that are commonly seen together (in other projects) are not present in a project. Such checkings are performed as an additional step after frame and transform types are inferred. The reason is that it is hard to implement implicit conventions as type rules as the mined conventions may change depending on the repositories used and the z-score threshold setting.

5 EVALUATION

We address the following questions: **RQ1.** How effective and efficient is PhysFrame in disclosing frame related inconsistencies? **RQ2.** What is the response from developers on the inconsistencies reported by PhysFrame? **RQ3.** What kinds of implicit conventions are violated and what is the impact of z-score on results?

5.1 Experiment Setup

Implementation. We have implemented a prototype of PhysFrame in around 4800 lines of Python code. It utilizes a third-party component, *cppcheck* [26], to obtain an XML dump for a C/C++ file providing an intermediate representation. It contains program tokens, symbol tables for scopes, functions and variables, and an AST for each statement. We have also implemented constant propagation to resolve variables to static values when possible. The miner is implemented using MySQL database and SQL script. Implementation details can be found in Appendix C in [19]. The database for the implicit convention miner is built from a list of ROS projects that contain at least one launch file with a *static_transform_publisher.* We use only the mature projects from this list. We consider a project mature when it has more than 30 commits. The database contains 12.1K static transforms in 5.3K launch files from 2.2K mature projects. The tool and data is available at https://doi.org/10.5281/zenodo.4959920

Artifacts. We run the type checker on 180 ROS projects from Github. They include (a) 7 projects from the ROS Kinetic distribution that contain at least one issue or a commit related to frame; (b) 23 randomly selected projects from the ROS indigo distribution; and (c) 150 randomly selected ROS projects that contain at least one static transform. Among these projects, 69 are mature and 99 not (with less than 30 commits). Note that we consider evaluation on unmature projects important as well because they are likely by non-expert ROS developers that can benefit more from PhysFrame. Table 7 in Appendix in [19] shows the statistics of these projects. On average, each project has 45 source/launch files and 52491 LOC, with the largest one having 433 files and over 7 million LOC.

Examination Process. We examine each of the reported inconsistencies by reviewing the corresponding source code and mark it as *True Positive* (TP) or *False Positive* (FP). The manual examination poses a threat to the validity of results presented in this paper (please refer to Appendix D in [19]). We mitigate it by cross-checking, being very conservative in marking TPs, and confirmation

Table 3: Inconsistencies by Category (with the first seven type errors, the last two type warnings)

Category	Number of Inconsistencies					Number of Repositories		
	Total (#)	TP (#)		FP (#)		Total (#)	TP (#)	FP (#)
		C/C++	Launch	C/C++	Launch			
c_MULTIPLE_PARENTS_IN_FRAME_TREE	4	3	1	0	0	4	4	0
c_CYCLE_IN_FRAME_TREE	1	1	0	0	0	1	1	0
td_INCORRECT_FRAME_ORDER_IN_TREE	7	3	4	0	0	7	7	0
c_INCORRECT_TRANSFORM	17	4	4	0	9	9	3	6
td_REDUNDANT_TRANSFORM	2	2	-	0	-	2	2	0
td_MISSING_FRAME	108	92	-	16	-	56	49	13
td_MISSING_CHILD_FRAME	12	8	-	4	-	7	4	3
td_REVERSED_NAME (type warning)	33	1	28	1	3	19	17	4
c_SENSOR_NULL (type warning)	6	0	3	0	3	5	2	3
Total	190	114 [81.05%]	40	21 [18.95%]	15	110	89	29

Table 4: Summary of True Positive Inconsistencies

Report & Response	Count	Inconsistency Categories
found by searching commit-history or issue reports of repositories.	3 1	td_MISSING_FRAME c_INCORRECT_TRANSFORM
fixed by developers before we reported them.	1 1 1 1	td_MISSING_FRAME td_REDUNDANT_TRANSFORM c_CYCLE_IN_FRAME_TREE c_MULTIPLE_PARENTS_IN_FRAME_TREE
reported, and either fixed by developers or ready to accept a pull-request.	2 1 1 1	td_MISSING_FRAME td_MISSING_CHILD_FRAME td_REDUNDANT_TRANSFORM td_INCORRECT_FRAME_ORDER_IN_TREE
reported, and acknowledged by developers (in some cases, confirmed as a useful fix to others for reusing the package node).	1 5 2 2	td_MISSING_FRAME td_REVERSED_NAME c_SENSOR_NULL td_INCORRECT_FRAME_ORDER_IN_TREE
reported, but refused by developers on the grounds that reuse of the package node is not easy, or the package is no longer maintained.	3	td_MISSING_FRAME
reported, but no response from developers yet.	12 1 4 14 1 2	td_MISSING_FRAME td_MISSING_CHILD_FRAME c_INCORRECT_TRANSFORM td_REVERSED_NAME td_INCORRECT_FRAME_ORDER_IN_TREE c_MULTIPLE_PARENTS_IN_FRAME_TREE
found in a package node that is reused by other developer. The issue is either fixed or acknowledged by other users in the original repository of the node.	2 4	td_MISSING_FRAME td_MISSING_CHILD_FRAME
found in backup-copy files in repositories.	5	td_MISSING_FRAME
potential issues.	8	td_MISSING_FRAME
projects not active in past two years.	55 2 3 10 1 3 1	td_MISSING_FRAME td_MISSING_CHILD_FRAME c_INCORRECT_TRANSFORM td_REVERSED_NAME c_SENSOR_NULL td_INCORRECT_FRAME_ORDER_IN_TREE c_MULTIPLE_PARENTS_IN_FRAME_TREE

Table 5: Implicit Convention Warnings by Category

Category	z=1		z=2		z=5		z=10	
	W#	R#	W#	R#	W#	R#	W#	R#
w_sig_null_rot_exp	18	9	16	7	14	5	9	2
w_name_null_rot_exp	20	10	20	10	2	2	0	0
w_name_co-occurrence	7	3	0	0	0	0	0	0

with developers. The implicit convention violation warnings are by their nature more difficult to determine if they denote real bugs. We hence run the tool on the same set of projects with different z-score threshold values and study the reported warnings. All type checking runs are on a Dell PowerEdge R420 Linux server with two Intel Xeon E5-2690 2.90GHz 8-core processors and 192 GB of RAM.

5.2 Results and Observations

RQ1. Effectiveness and Efficiency. PhysFrame reports 190 inconsistencies with 154 true positives and 81.05% true positive rate. These inconsistencies belong to 100 projects, over 55% of the total we examined. This number is specially significant considering that many more frame related bugs reported in earlier versions of the systems may have been detected by PhysFrame. PhysFrame also reports 45, 36, 16, 9 implicit convention violation warnings when the z-score threshold is set to 1, 2, 5, 10, respectively, which corresponds to p-values smaller than 0.15 for z=1 to p-values smaller than 0.00001 for z=10. The average analysis time is 52 seconds with the maximum 1190 seconds. Further details can be found in Table 6 in Appendix in [19].

Inconsistencies by Category. Table 3 summarizes inconsistencies by their category, with the first seven type errors and the last two type warnings. The first column lists the categories. Each has either a '*c_*' or '*td_*' prefix, denoting whether it is a "correctness" or a "technical debt" issue, respectively. The "correctness" issues represent the types of inconsistencies that violate ROS conventions such as a tree representation of all the frames in an application, sensor frames should have non-null transforms with respect to robot body frames, and incorrect transforms. The incorrect transform issues include those that have invalid displacement/rotation (like our motivating example in Section 3) which can lead to problems during runtime, and those that do not follow ROS naming conventions (discussed in Section 2.2) while defining a child frame in the transform which affect code readability and may cause issues during code reuse. The "technical debt" issues represent the types of inconsistencies that may make code maintenance and understanding difficult and may lead to potential runtime problems during code reuse/collaboration. The next five columns show the total number of inconsistencies for each category, TP cases in C/C++ code and in launch files, FP cases in C/C++ code and in launch files, respectively. Further, the last three columns show the number of projects with inconsistencies, TPs, and FPs, respectively.

Many of the reported inconsistencies are about missing frames, i.e., developers forgot to set frame ids for published state values. FP cases of this type are observed due to usage of stamped data types for storing non-frame data (e.g., binary mask of image), usage of stamped transform for tf operations that do not need *frame_id* field value, and passing of stamped data type variables to functions whose signature is not available during analysis. Similar sources of false positives affect other categories. PhysFrame also found many reversed name warnings. For example, ROS expects transform variables to follow the naming convention of "⟨parent⟩_to_⟨child⟩". We find that many developers use the names reversed. This may cause maintenance and reuse issues as the convention is expected by other parties. PhysFrame also identified 17 frame type inconsistencies similar to our motivating example in Section 3 and 12 TF tree errors. Given that ROS is built to encourage collaborative development, even the issues affecting code readability or reuse can be important. We have three case studies in Appendix E in [19].

RQ2. Developer Responses. Table 4 summarizes the distribution of the true positives across different reporting actions, developer responses and inconsistency categories. Row 1 (below the table title) shows that 4 of the 154 true positives PhysFrame uncovered were known issues by the developers. Row 2 shows that another four were fixed by the time we reached the developers. Among the remaining true positives, we reported the 52 that belong to projects that showed activity in the past 2 years (rows 3-6). We received 18 responses with 15 either acknowledging this issue or fixing it, and 3 declining our reports because either it is not seen as relevant (project is not intended for external use hence conventions do not need to be respected), or project no longer maintained. Row 8 shows that PhysFrame found 6 bugs in reused versions of projects that have already been fixed in the original repositories.

RQ3. Implicit Convention Violations. Table 5 shows the three kinds of implicit convention violations PhysFrame has found and the impact of z-score threshold. W# means the number of warnings and R# the number of projects with warnings. The most popular kind is *w_name_null_rot_exp*, meaning that a transform with a specific name is expected to have 0 rotation; *w_sig_null_rot_-exp* is similar except that a transform is identified by parent and child frame ids instead of name. Some of the *w_sig_null_rot_exp* warnings are persistent (with different z-threshold), meaning that conventions adopted by almost all projects are being violated.

6 RELATED WORK

Component-based Robot Software. Component-based programming [8, 9] has been extensively used in robot development, aiming to facilitate reuse. Following the principle, several robot software frameworks [25], such as Open Robot Control Software (Orocos) [10], ROS [32]. Player [17], Miro [37], have been developed and widely used. These frameworks commonly provide communication infrastructure across components [16] and substantial library support. As such, respecting conventions is critical for these infrastructures. Unfortunately, they provide no compile time support to ensure frame conventions, one of the most error-prone aspects of robot programming. PhysFrame is a type checking technique that systematically detects problems in using frames.

Type Checking in Robotic Systems. Type checking [11, 33] is a well-known technique to improve software reliability. In robotic software, due to its domain-specific characteristics, variables often have physical-world semantics. Additional type checking can hence be performed considering the physical world. Unit type checking [7, 12] is the most common robotic software type system, which type variables with physical unit information. The seminal work [21, 22, 34] in the type checking of units and dimensions extends a programming language. Ayudante [18] builds and compares clusters of variables based on dataflow and the meaning of variables names to detect type (e.g., unit) inconsistencies. The recent work on Phriky [30, 31] supports dimensionality type checks in ROS without developer annotations, and Phys [20] extends it with probabilistic type inference to detect more inconsistencies. In this work we extend the type checking to reference frames in robotic software, which requires a distinct type system.

Frame Safety. Lowry et al. [23] proposed a general abstract interpretation system for certifying domain-specific properties and provide a case study to certify NAIF library programs' frame-safety. The case study checks limited properties (e.g., frame consistencies) and the system needs annotations for program inputs. In contrast, ours is a fully automated tool checking many properties for ROS projects (e.g., TF tree shape and frame orientations), without requiring annotations. Our abstract domains are hence different from theirs and our method is type system based. The work by Basir et al. [6] on automated theorem proving for automatically generated code also presented a case study to verify two frame-safety requirements. META-AMPHION system [24] helps domain experts build domain-specific program synthesizers. In the context of frame-safety, we believe that this work would enable the synthesis of programs that are frame-safe given the correct domain theory. In our work, we focus on finding frame issues in the already developed code. Overall we do not need annotations, have a push-button approach, and provide a broad assessment of real code.

7 CONCLUSION

We develop a fully automated type checking technique for reference frames in robotic system, which is one of the most subtle and error-prone aspects in robotic software engineering. We explain the semantics of frames and frame transformations, and propose a novel way to abstract them. Type checking rules are developed based on the domain specific semantics. Applying our technique to 180 ROS projects from GitHub discloses over 190 errors and warnings, with 154 true positives. We reported 52 of them and received 18 responses by the time of submission and 15 fixed/acknowledged.

ACKNOWLEDGMENTS

We thank our insightful reviewers. This research was supported, in part by NSF 1901242, 1910300, 1909414 and 1853374, ONR N0001417-12045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2015. *diff_drive_controller is hard-coded into the 'odom' reference frame #204.* https://github.com/ros-controls/ros_controllers/issues/204

[2] 2017. *TF: Debugging Tools.* http://wiki.ros.org/tf/Debuggingtools

[3] 2019. *ORB-SLAM2 ROS node.* http://wiki.ros.org/orb_slam2_ros

[4] 2019. *T265 frame questions #772.* https://github.com/IntelRealSense/realsense-ros/issues/772

[5] 2020. *Incorrect frame tree cartographer ros and t265 camera #1599.* https://github.com/IntelRealSense/realsense-ros/issues/1599

[6] Nurlida Basir, Ewen Denney, and Bernd Fischer. 2010. Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security* (Vienna, Austria) *(SAFECOMP'10)*. Springer-Verlag, Berlin, Heidelberg, 68–81.

[7] Geoffrey Biggs. 2007. *Designing an application-specific programming language for mobile robots.* Ph.D. Dissertation. ResearchSpace@ Auckland.

[8] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback. 2005. Towards component-based robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 163–168.

[9] Davide Brugali and Patrizia Scandurra. 2009. Component-based robotic engineering (part i)[tutorial]. *IEEE Robotics & Automation Magazine* 16, 4 (2009), 84–96.

[10] Herman Bruyninckx. 2001. Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, Vol. 3. IEEE, 2523–2528.

[11] Luca Cardelli. 1996. Type systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 263–264.

[12] Kostadin Damevski. 2009. Expressing measurement units in interfaces for scientific component software. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*. 1–8.

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[14] Tully Foote. 2013. tf: The transform library. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA), 2013 (Open-Source Software workshop)*. 1–6. https://doi.org/10.1109/TePRA.2013.6556373

[15] Tully Foote and Mike Purvis. 2010. *Standard Units of Measure and Coordinate Conventions.* https://www.ros.org/reps/rep-0103.html

[16] Open Source Robotics Foundation. [n.d.]. *ROS Topics homepage.* http://wiki.ros.org/Topics

[17] Brian Gerkey, Richard T Vaughan, and Andrew Howard. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, Vol. 1. 317–323.

[18] Irfan Ul Haq, Juan Caballero, and Michael D Ernst. 2015. Ayudante: Identifying undesired variable interactions. In *Proceedings of the 13th International Workshop on Dynamic Analysis*. 8–13.

[19] Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian Elbaum. 2021. PHYSFRAME: Type Checking Physical Frames of Reference for Robotic Systems. arXiv:2106.11266 [cs.RO]

[20] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. 2018. Phys: probabilistic physical unit assignment and inconsistency detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 563–573.

[21] Andrew Kennedy. 1994. Dimension Types. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems (ESOP '94)*. Springer-Verlag, Berlin, Heidelberg, 348–362.

[22] Andrew Kennedy. 2009. Types for Units-of-Measure: Theory and Practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School* (Budapest, Hungary) *(CEFP'09)*. Springer-Verlag, Berlin, Heidelberg, 268–305.

[23] Michael Lowry, Thomas Pressburger, and Grigore Rosu. 2001. Certifying Domain-Specific Policies. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*. IEEE Computer Society, USA, 81.

[24] Michael R. Lowry and Jeffrey Van Baalen. 1997. META-AMPHION: Synthesis of Efficient Domain-Specific Program Synthesis Systems. *Automated Software Engg.* 4, 2 (April 1997), 199–241. https://doi.org/10.1023/A:1008637201658

[25] 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.

[26] Daniel Marjamaeki. 2017. *Cppcheck - A tool for static C/C++ code analysis.* http://cppcheck.sourceforge.net/

[27] Wim Meeussen. 2010. *Coordinate Frames for Mobile Platforms.* https://www.ros.org/reps/rep-0105.html

[28] Thomas Moulard. 2011. *Coordinate Frames for Humanoid Robots.* https://www.ros.org/reps/rep-0120.html

[29] Raúl Mur-Artal and Juan D. Tardós. 2017. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103

[30] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Lightweight Detection of Physical Unit Inconsistencies Without Program Annotations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 341–351. https://doi.org/10.1145/3092703.3092722

[31] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 352–355. https://doi.org/10.1145/3092703.3098219

[32] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3.2. Kobe, Japan, 5.

[33] John C Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*. Springer, 408–425.

[34] Mikael Rittri. 1995. Dimension Inference under Polymorphic Recursion. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 147–159. https://doi.org/10.1145/224164.224197

[35] Clearpath Robotics. 2017. *Robots - PR2.* http://wiki.ros.org/Robots/PR2

[36] Wim Meeussen Tully Foote, Eitan Marder-Eppstein. [n.d.]. *Ros tf package homepage.* http://wiki.ros.org/tf,http://wiki.ros.org/tf2

[37] Hans Utz, Stefan Sablatnog, Stefan Enderle, and Gerhard Kraetzschmar. 2002. Miro-middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation* 18, 4 (2002), 493–497.