Linting *roslaunch* Static Transforms

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Michael Chinn

Spring, 2020

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Sebastian Elbaum, Department of Computer Science

Linting *roslaunch* Static Transforms

Michael Chinn
Department of Computer Science
University of Virginia
Charlottesville, USA
mec2wr@virginia.edu

I. Abstract

In this research we identified and classified antipatterns in the specification of static frame transformations in the Robot Operating System (ROS) framework. We determined these antipatterns using a dataset composed of static transforms from public GitHub repositories. We built Launch-Linter, a linter which detects the presence of these antipatterns in *roslaunch* configuration files. We executed Launch-Linter on a subset of files from GitHub repositories which were not used to determine the antipatterns and found errors in 9.6% of files tested.

II. Introduction

Robotics systems are becoming increasingly used in different industries. Ensuring the reliability of the robot software is critical, as faults may result in personal injury or expensive crashes. In this paper we explore the space of faults that arise due to incorrect transformations among frames of references in robotics systems, and design automated mechanisms to detect those faults.

A frame of reference is a coordinate system which can be used to describe the position and orientation of objects. When working with robotics systems, there is often a *world reference frame* that is used to specify the location of the system [1]. Additionally, there may several meaningful reference frames within a robot to specify the relative position of system components. These frames of reference provide an abstraction allowing software developers to more easily conceptualize and transform the locations of the different components of a robot as they interact.

For instance, for a robotic arm, there may be a shoulder joint, an elbow joint, and a wrist joint. The shoulder may be fixed relative to the world frame; the position of the elbow relative to the ground is based upon the position of the elbow relative to the shoulder, and the position of the wrist relative to the ground depends on the position of the elbow relative to the shoulder. In cases like this, developers must explicitly convert between reference frames.

Figure 1 shows an example of how reference frames might be setup for a robotic arm. Each set of red, green, blue markers defines a frame of reference, where the red marker corresponds to +X, the green marker corresponds to +Y, and the blue marker corresponds to the +Z direction. Between each reference frame there must be a transformation to relate how each frame is positioned and oriented relative to the other frames. There is a natural cascading effect where the transform from the end of the arm to the base of the arm can be composed by applying each intermittent transformation between the two in series.

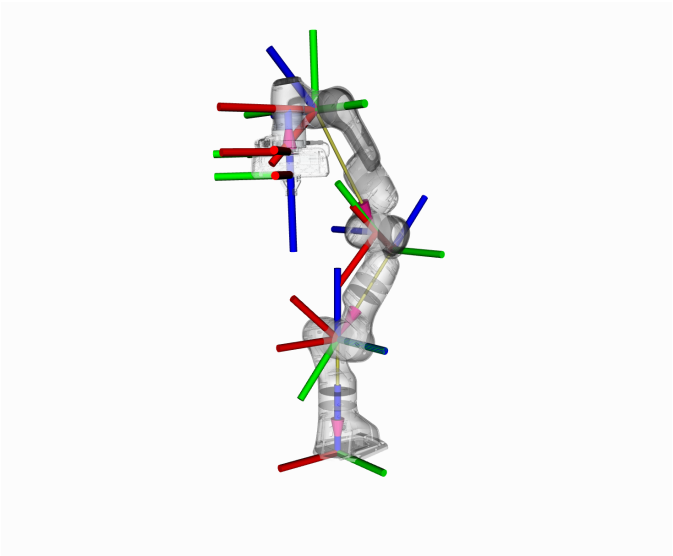

Fig. 1. Example Reference Frames
Taken from this GitHub repository:
https://github.com/ros-planning/moveit_tutorials, released under a BSD license.

We consider the usage of frame transformations in the Robot Operating System (ROS), a very commonly used framework for developing robotics software. ROS provides several mechanisms for specifying the reference frames that compose a robotics system, one of which involves specifying transformations in *roslaunch* files, which are configuration files structured like XML. We focused on this method of specifying transformations in our research.

In this research we: 1) identified three kinds of antipatterns in the definitions of these transformations that will or may lead to faults, 2) introduce a specialized linter which pinpoints the presence of these antipatterns in *roslaunch* files before deployment, and 3) evaluated our linter on the transforms present in immature repositories to determine how commonly the errors occur. We found errors in 9.6% of the nearly 8500 files we used in evaluation.

III. Background

A. Static Transformations in ROS

The Robot Operating System (ROS) is a popular framework for developing robotics software. ROS provides the *tf* package for automatically handling transformations based upon developer specifications. This package was created to make declaring transforms simpler and less prone to errors [2].

A subset of *tf* transforms are static and cannot change over time. These may be declared within *roslaunch* XML configuration files using a *static_transform_publisher*. Displacement and rotation parameters are taken as whitespace separated, untyped arguments, so it is challenging for developers to compose error-free static transforms.

Fig. 2 shows an example of a *static_transform_publisher*. The relevant attributes are its *name* and its *args*. The *name* refers to the *static_transform_publisher* and the *args* are a whitespace separated list of parameters. The first three parameters are numeric and refer to the X, Y, and Z values for the displacement of the transformation. These may be followed by either three or four additional numeric arguments which describe the transformation rotation. If there are three arguments, they will be interpreted as the yaw, pitch, and roll (YPR) values in radians. Otherwise the four arguments define the rotation as a quaternion and are q_x, q_y, q_z, and q_w. After the numeric arguments are two strings. The first of these is the name of the parent frame and the second is the name of the child frame. The transformation goes from the parent frame to the child frame. The numerical arguments to a transform may all be zero, in which case the transform exists to map the child frame name to the parent frame and we refer to the transform as a null transform. The displacement and rotation arguments may independently be null based on the purpose for the transform.

```
<!-- STATIC TRANSFORM FROM DRONE BODY TO ToF SENSOR 1-->
<node pkg="tf"
      type="static_transform_publisher"
      name="drone_ToF_sensor_1"
      args="0 0.55 0 1.3 0 0 /firefly/base_link /quat/tof_tf_1 100"/>
```

Fig. 2. Sample *static_transform_publisher*

We will often refer to the combination of parent and child frame names as the signature of a transformation, and we may also write a transformation in the form (*parent_name* $\rightarrow$ *child_name*).

roslaunch files additionally provide a mechanism for performing macro expansion using arbitrary python expressions, so the full attributes of a *static_transform_publisher* may not be known statically. An example of this is shown using the 'eval' construct in Figure 3.

```
<node pkg="tf2_ros" type="static_transform_publisher" name="$(anon tf_static_world_to_map)"
    args="$(eval ' '.join([str(map_origin_x), str(map_origin_y), '0 0 0 0 world map']))"/>
```

Fig. 3. Static Transform Using Eval

These configuration files are often shared between projects, so errors made in the declaration of a single transform may propagate widely. Of the approximately 500K *roslaunch* files we queried from public GitHub repositories, about 55K contain a *static_transform_publisher*, and we roughly estimate that about a third of them are duplicates.

```
    <!-- Specifying that the laser is rotated 135 degrees by x relative to
the center of the robot -->
    <node pkg="tf" type="static_transform_publisher" name="base_to_laser_tf"
  args="0 0 0 135 0 0 /base_link /laser_frame 50" />
```

(a) Wrong rotation unit

```
<node pkg="tf" type="static_transform_publisher" name=
    "static_transform_publisher1" args="0.0 0.365 0.2 -1.57 0.0 0.0
    base base_wheel_left"/>
<node pkg="tf2_ros" type="static_transform_publisher" name=
    "static_transform_publisher1" args="0.0 -0.365 0.2 -1.57 0.0
    0.0 base base_wheel_right"/>
<node pkg="tf2_ros" type="static_transform_publisher" name=
    "static_transform_publisher3" args="0.0 -0.365 0.2 -1.57 0.0
    0.0 base base_wheel_center"/>
```

(b) Duplicated name, missing integer suffix

```
<node pkg="tf" type="static_transform_publisher" name="map_to_odom" args="0.0 0.0
  0.0 0.0 0.0 0.0 /map /base_footprint 10"/>
<node pkg="tf" type="static_transform_publisher" name="base_footprint_2_base_link"
  args="0.0 0.0 0.0 0.0 0.0 0.0 /base_footprint /base_link 10"/>
<node pkg="tf" type="static_transform_publisher" name="base_link_2_camera_link"
  args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /camera_link 10"/>
<node pkg="tf" type="static_transform_publisher" name="camera_depeth_frame_2_nav"
  args="0.0 0.0 0.0 0.0 0.0 0.0 /camera_depth_optical_frame /nav 10"/>
```

(c) Name implication violated

Fig. 4. Motivating Examples

B. Motivating Examples

To concretize the kinds of errors present in *roslaunch* static transforms, Figure 4 shows some motivating error examples found in public GitHub repositories.

(a) shows a transform which consists of a 135° rotation. This is a fault, as the *static_transform_publisher* interprets its arguments in radians, not degrees.

(b) shows a set of three static transforms. Two of the transforms are named *static_transform_publisher1* and the third is named *static_transform_publisher3*. The duplicated name may lead to developer confusion as only the first one would be instantiated, but it is also apparent that a transformation is missing given that the integer suffixes 1 and 3 being present but not 2, so the missing name may be *static_transform_publisher2*.

(c) shows an error of a very different variety. It shows a collection of four static transforms which are present in a file, one of which is named *base_footprint_2_base_link*. From examining thousands of transforms in public repositories, we have determined that there is a commonly followed developer convention that when a transform with that name is present there should also be a transform named *base_2_nav_link* present. As can be seen, there is no transform with that name. This error of omission is a violation of an implicit developer convention and may lead to challenges when collaborating or maintaining this software.

IV. Identifying Faulty Patterns in Static Transforms

The goal of this work is to ultimately develop rules that represent antipatterns in static transforms that arise due to the flexibility of the *roslaunch* configuration files, which impose few limitations to the kinds of names and values which may be used.

Our antipatterns are based on two sources.

First, explicit semantic specifications for the transformations. Violating these specifications render certain or likely invalid or incorrect transformations, such a specifying a rotation parameter in degrees when radians is expected.

Second, most developers encode beliefs in the labels and values they provide in their transforms, which indicate their expectations of how the code functions. When developers break these shared beliefs their code may still function, but collaboration and maintenance will both be more challenging. Identifying such beliefs can be valuable as this information may not actually be documented anywhere.

We consider a code pattern to be a combination of multiple configuration properties which may occur at the scope of a single transform or at the scope of an entire file of transforms. A code pattern may represent a certain combination of frame names, the correspondence of a transform name to the values of the transform, or the existence of certain transforms within the same file. When each property of a code pattern is met, we consider the pattern followed. If at least one property of the pattern is met but the condition is not followed, we consider the pattern broken.

Code patterns may be followed purely by coincidence, but when a pattern is followed frequently we believe it may represent a developer's belief that the component properties should be followed together. In this case, violations of the the pattern should be linted for as breaking it would violate developer conventions and may result in a fault or otherwise impede developer collaboration or the maintainability of the software. When a pattern is broken in a file we think it is less likely that the pattern represents a rule that must always be followed, although the breaking of the pattern may itself be a fault.

A. Rule Development Process

The development of rules followed an iterative process. We began with hypotheses grounded in our knowledge about static transform usage and likely faults we made or that we found as issues or questions in ROS repositories. We then collected data to support, refute, or refine the hypotheses, and we employed quantitative measures to validate the hypothesis. At each stage in this process we employed the additional knowledge we gained to generate new hypotheses. Figure 5 is a graphical representation of this process. We will now describe the process in more detail.

1) Bootstrapping Initial Hypotheses: We started by becoming more familiar *tf* usage and misusage by other developers to gain a better understanding of what kinds of beliefs are implicit in configuration files. We searched public GitHub repositories

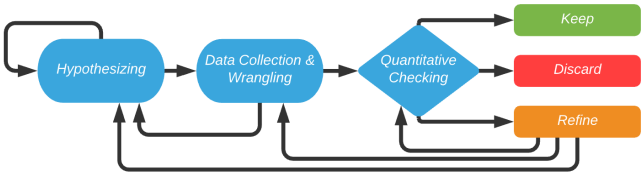

Fig. 5. Rule Generation Approach

for instances of developers using *static_transform_publisher* and examined over a hundred comments, issues, and commit histories to see how developers corrected mistakes over time.

From this examination we learned, for example, that transform arguments often included uncommented constants for both the displacements and the rotations arguments. We also noted that while the changes were rarely documented, they often consisted of flipping the sign of numerical arguments or permuting their order. We also found that when files do have many static transforms, they often took the form of inscrutable blocks of code, as in Figure 6 which seemed susceptible to copy-and-paste errors resulting in propagating errors in code. Most important, we also frequently saw the same frame names and many values appearing in different *roslaunch* files, giving us the first hint that developers do encode implicit beliefs in their configuration files. With that in mind, we began to establish a more formal approach to examining static transforms.

```
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_62"
args="6.045 7.905 0 1.57 0 0 map tag_62" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_53"
args="5.115 7.905 0 1.57 0 0 map tag_53" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_44"
args="4.185 7.905 0 1.57 0 0 map tag_44" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_35"
args="3.255 7.905 0 1.57 0 0 map tag_35" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_26"
args="2.325 7.905 0 1.57 0 0 map tag_26" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_17"
args="1.395 7.905 0 1.57 0 0 map tag_17" />
  <node pkg="tf2_ros" type="static_transform_publisher" name="map_tag_8"
args="0.465 7.905 0 1.57 0 0 map tag_8" />
```

Fig. 6. Block of Static Transforms

One of our initial bootstrapped hypotheses was that certain certain numerical arguments to transforms would appear together. In this informal process, we found that the arguments tended to consist of magic constants which would appear without comment among transforms. We were unable to relate the values of the numerical arguments to anything else in the code so we abandoned this hypothesis at this point. Another one of our hypotheses was that some transform signatures always or often correspond to null transforms. This hypothesis was strengthened during this informal analysis, so we examined it further when we employed a more data-driven approach.

2) Data-Driven Approach: With a better understanding of use cases for the *static_transform_publisher* we implemented a pipeline for building a dataset of ROS transforms. We could then construct queries representing code conditions to count the frequencies of the condition being satisfied and being

broken. This system allowed for much more rapid iteration and formalization of our hypotheses and error patterns.

The pipeline included a large dataset of *roslaunch* files from public GitHub repositories. Using the GitHub code search API, we downloaded each of the 55K launch files which contained a *static_transform_publisher*. For each of these files we used the GitHub API to download the commit history of the repository, and if the launch file we downloaded had been modified in a commit, we additionally downloaded the original file version.

We began by removing files which could not be parsed correctly. As mentioned previously, many of these files were repeated in different repositories as they were shared. All duplicated files were removed.

We created a MySQL database where each row represented a single transform. The fields included each of the attributes and arguments for the transform, in addition to a file id, a repository id, a flag indicating if the file was the original commit or a later revision, and a flag indicating if the repository had a least 30 commits (which we deemed as more mature).

For each of the files (new and old) that we had downloaded, we parsed the XML specification and added its *static_transform_publisher*s into the SQL table. We skipped transforms that employed macro expansion, as these could not easily be separated by whitespace to determine the arguments.

In total, our dataset contained 26K static transforms defined in 13K files. These files came from 6K different repos.

Using this dataset we were able to easily run queries using potential error antipatterns to determine how frequently the pattern is followed and violated.

At this stage, an example of a hypothesis we had was that when there was a transform name that included the word "left", there would be a corresponding transform with the word "right" to represent symmetries of autonomous systems. Using the dataset we surprsingly found that transforms including the words "left" and "right" were not as common as we expected, so we chose not to continue focusing on this hypothesis.

An additional hypothesis we were able to better explore using our dataset was that when a transform's name indicated that it corresponded to the placement of a sensor it would not be a null transform. Using our dataset we were able to visualize and quantify the distribution of numerical arguments which corresponded to the displacement and rotations of transforms which included sensor names in the frame names or transformation name. We found evidence to support this hypothesis, and we included an error pattern based upon it in our final linter.

3) Quantifying the Significance of Rules: The data analysis we performed led to some interesting questions, such as: are transforms with certain names meant to be non-null, or does the existence of a transform with a certain signature in a file imply that a transform with another signature should also be present? These kinds of questions are unlikely to have strict documented rules which are followed, as the *roslaunch* configuration system is flexible, and developers may use the same names differently. However, when a large portion of files do follow the same convention, despite functioning software, it may be an antipattern to violate the convention as it will make maintenance and collaboration more challenging and error prone.

To quantitatively study these kinds of implications, we used a method involving MAY beliefs and MUST beliefs, used in testing systems code in [3]. A MUST belief is a code pattern which always must hold and a MAY belief is any observed code pattern that may be a condition which must not be violated, but may also be a coincidentally observed pattern. Any time a MAY belief pattern appears in source code this contributes evidence that the MAY belief must hold, while any time it is violated contributes evidence that the condition is just a coincidence and breaking it is not harmful. Importantly violations of the MAY belief conditions do not always mean the MAY belief is incorrect, as the violations may themselves be errors since they are taken from existing source code. To then gauge how likely it that a MAY belief should be treated as a MUST belief, researchers in [3] propose a z score:

$$z(n, e) = (e/n - p_0)/\sqrt{(p_0 * (1 - p_0)/n)}$$

where n is the number of times the condition is checked, e is the number of examples checked which satisfy the condition, and p_0 is the expected probability that the condition will be satisfied when it is checked. They assume $p_0 = 0.9$ and note that z grows as n increases and $n - e$ decreases. A threshold value t is then applied based to the z scores, and any MAY belief with $t > z$ is treated as a MUST belief.

B. Validated Hypotheses: Rules We Check

We classified the error patterns we uncovered using the dataset into three categories: MUST errors, MAY-Semantic errors, and MAY-Implied errors. MUST patterns are those which are never correct in practice. MAY-Semantic errors are antipatterns which violate developer beliefs encoded in configuration files with sufficient underpinning logic to justify always checking for them. MAY-Implied patterns are those which are taken entirely from encoded developer beliefs and are determined using the z score method previously discussed.

The MUST errors are enumerated and described in Table I. These errors were determined to always be faults via the documentation for *tf*, which disallows them.

The MAY-Implied errors we considered are listed in Table II. These correspond to the hypotheses we developed based on the semantics of the transform arguments, and confirmed confirmed by the collected data.

The semantics for each table entry is as follows:

1) *reversed_name* refers to transforms with names which suggest that they refer to the reverse of the transformation to which they actually apply. ROS static transforms go from the parent frame to the child frame. An example violation of this rule is a transform named *map_to_odom* with the parent frame *odom* and the child frame *map*.
2) *rot_degrees* refers to transforms with a YPR rotation format with values that appear as degrees. In performing min/max analysis of numerical parameters for transforms we found YPR values as high as 135. Given that

TABLE I
STATIC TRANSFORM MUST ERRORS

Name	Description	Justification
self_transform	The same string is supplied for the parent and child frame arguments	This transform is nonsensical. The developer likely included the wrong parent or child so the behavior will not match expectation
dup_name	The same string is supplied as the transform name for two transforms in the same file	This will make modifying code challenging as the meaning of the transforms will not be clear
dup_sig	The same parent, child frame pair is used for two transforms in the same file	The pair of transforms may be contradictory and behavior will not match expectations

TABLE II
STATIC TRANSFORM MAY-SEMANTIC ERRORS

Name	Description	Justification
reversed_name	The transform name implies that it is from the child to the parent, rather than from the parent to the child. $[name] = [child][\|\_\|\_to\_\|2\|\_2\_][parent]$	The transform may be the inverse of what it is meant to be. The name does not reflect the code's function and will make modification challenging
rot_degrees	The value of a YPR rotation suggests it is in degrees rather than radians	The transform will not have the effect desired
int_suffix	If there are several frames with integer suffixes but an intermediate suffix is missing, this may be an error	The frame may have been skipped over in a list
ned_transform	If the child of a transform is suffixed with _ned, it should have a null displacement and a non-null rotation	This suffix is meant to be used when moving a coordinate system into NED according to REP 103. Displacement is not necessarily wrong, but it was not found in any transform from a mature repo within my dataset
sensor_null	A sensor transform (velodyne, kinect, openni, imu) is likely to include a small nonnull transform	Large values seem implausible for appropriate usage of most robotic sensors. A null transform would not do very much and it seemed uncommon to use a sensor name if just remapping a frame name

these parameters are in radians, it was surprising to find values greater than 2π.

3) *int_suffix* refers to transformation names or frame names which end with an integer suffix. In examining aggregated names from the dataset, we found that it was common to see files with similarly named and suffixed transforms to refer to related entities or sensors. Figure 2 shows an example of this kind of transform that appeared in a list of time-of-flight sensors on a drone.
4) *ned_transform* refers to child frame names with the suffix "_ned". ROS Enhancemant Proposal (REP) 103 which contains ROS standards for units and coordinate conventions states that ROS coordinate frames should follow the East North Up (ENU) convention, and that when a frame is needed to follow the North East Down (NED) convention, as is more typical for systems operating outdoors some frame should be transformed to a child frame with the "_ned" suffix. From this convention, this transformation should generally always include a rotation and involve no displacement. From the dataset we confirmed that these transforms very rarely have non-null displacement and nearly always have non-null rotations.
5) *sensor_null* refers to transformations where the name, parent frame name, or child frame name contains the words velodyne, kinect, openni, or imu, all of which refer to sensors. As mentioned previously, a transform may be completely null, in which case it just serves to map two names to the same frame. While this may be used generally as a developer convention, it seemed unlikely that sensors would have null transforms, as these likely refer to physical devices in a robotics system. We confirmed through our dataset that transforms with these names were generally non-null.

Finally, MAY-Implied errors are errors which violate patterns which are determined using the previously discussed z score method. These patterns are described in Table III. Each entry in the table encompasses a class of rules which follow the given pattern. For each pattern in the class, we followed the previously described quantification process to assign a z score (using $p_0 = 0.9$) to indicate how likely it is that violations of the pattern are actually errors. For example, *sig_implication* refers to patterns of the form $\exists(A \to B) \implies \exists(A' \to B')$ where A, A', B, and B' are frame names. An instance of this pattern would be $\exists(base\_stabilized \to base\_frame) \implies \exists(base\_link \to base\_stabilized)$. A transform with the signature $(base\_stabilized \to base\_frame)$ appears in 42 files from mature repositories. In every single one of those files there is also a transform with the signature $(base\_link \to base\_stabilized)$. The z score is then given by $z(42, 42) =$

$(42/42-0.9)/\sqrt{(0.9*(1-0.9)/42)} = 2.16$, which indicates a strong likelihood that developers believe these two transform signatures belong together, so this instance of the more general pattern is included as a MAY-Implied pattern in our linter.

V. Linter Generator

From our dataset of static transformations we identified three categories of transformation antipatterns. The first two of these are the MUST errors from Table I and the MAY-Semantic errors from Table II, and these conditions are fixed. The third category of errors are derived from the MAY-Implied patterns in Table III. These are heavily dependent upon the generated dataset to determine a z score for each rule following the pattern and the threshold values chosen to distinguish for which instances of the MAY-Implied patterns violations are treated as antipatterns. Given that the dataset these beliefs are based upon is subject to change, we did not create a fixed linter for *roslaunch* configuration files.

Instead we created a linter generator which takes in a set of MySQL implication tables (of rules within the MAY-Implied pattern classes and their z scores) along with corresponding threshold values and uses a Jinja template to create a Python linter based upon the selected implications. Figure 7 represents the usage of this linter.

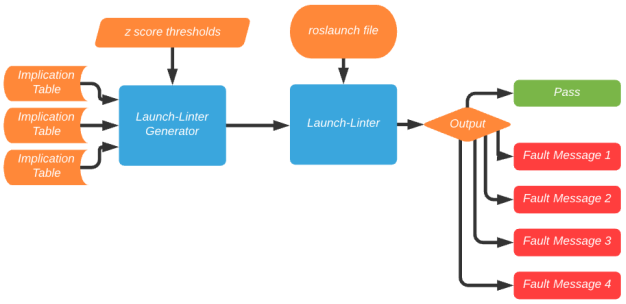

Fig. 7. Launch-Linter Usage

A. Applying the Launch-Linter

We executed Launch-Linter on each of the *roslaunch* files in our dataset which came from repositories with fewer than 30 commits. These were deemed to be the less mature repositories and they were excluded from the generation of the MAY-Implied beliefs.

Launch-Liner was run on 8438 files. Our linter indicated errors in 811 files, which corresponds to a 9.4% error rate. The full distribution of these errors is shown in Table IV. Some of the errors which were found are clearly incorrect; many of the 86 detected among the *self_transform*, *dup_name*, and *dup_sig* are clearly incorrect and confirmed by inspection (though some are not faults due to macro expansion, as discussed in the section on limitations below). Other errors matching the *rot_degrees*, *int_suffix* were manually confirmed to be incorrect configurations. Additionally the *reversed_name* errors were confirmed to break standard developer conventions, and

```
<node pkg="tf" type="static_transform_publisher" name=
"base_link_2_base_stabilized_link" args="0 0 0 0 0 0 /base_link /_base_stabilized
100"/>
<node pkg="tf" type="static_transform_publisher" name="base_2_nav_link" args="0 0
0 0 0 0 /base_frame /nav 100"/>
```

(a) MAYBE Fault Correctly Identified

```
<node pkg="tf" type="static_transform_publisher" name="stp_laser" args="0.2 0
0.08 3.14159 3.14159 0 base_link horizontal_laser_link 1" />
<!-- <node pkg="tf" type="static_transform_publisher" name="stp_laser"
args="0.285 0 0.24 0 0 0 base_link horizontal_laser_link 1" /> -->
```

(b) MAYBE Fault Incorrectly Identified

Fig. 8. Examples of MAYBE Faults Identified

while these may not be faults, they will certainly impair the maintainability of the software and make collaboration more challenging.

The errors which are most challenging to manually confirm as incorrect are those which match the patterns for MAY-Implied errors. Developers have wide latitude in their choice of names, and though the MAY-Implied errors indicate a more widely used convention, the *roslaunch* may not exhibit any faults. Furthermore, since these selected rules are selected based on their z score, the choice of cutoff threshold will impact whether these errors are over or under reported.

Figure 8 shows examples from two files that the Launch-Linter indicated exhibited MAY-Implied errors. (a) was flagged with the message that the rule $(\textit{base\_frame} \rightarrow \textit{nav}) \implies (\textit{base\_link} \rightarrow$ base_stabilized) was violated. While at a glance it appears that the code in (a) does not violate this rule, the first transform actually has the signature $(\textit{base\_link} \rightarrow$ _base_stabilized) where the child frame has a leading underscore. When using *tf* within ROS source code, frames are referred to by string names. If in another place the developer follows the convention of not using the leading underscore, or if the code must interact with a library which assumes the presence of the frame, the leading underscore will lead to interoperability challenges. It is easy to miss the presence of the underscore in the large body of configuration in the file from which (a) was taken. (b) was flagged for violating the rule $T_{name} = \textit{stp\_laser} \implies T_{rot} = \vec{0}$. This violation is unlikely to represent a fault, as it can be seen in the file's comment that the present transform with non-null rotation replaces a transform which had a null rotation. Since the developer originally followed the rule and consciously modified the software, it is unlikely that this rule was incorrectly broken.

B. Linter Limitations

There were several limitations of Launch-Linter to checking existing *roslaunch* files.

1) When Launch-Linter parses *roslaunch* files, it ignores transforms which use runtime macros to populate arguments. This may cause the linter to over report errors which occur at the file scope, as the linter will not account for the name or signature of the transforms with macros. Additionally, if macros are used outside of specific transforms to choose which are present at

TABLE III
STATIC TRANSFORM MAY-IMPLIED PATTERNS

Name	Scope	Description	Specification
sig_implication	File	A child, parent transform pair implies the existence of second child, parent pair	$\exists(C \to P) \implies \exists(C' \to P')$
name_implication	File	A transformation with a given name implies the existence of second transformation with a given name	$\exists T : T_{name} = N \implies \exists T' : T'_{name} = N'$
sig_null_disp	Transform	Some signatures imply that the transform should have null displacement	$(C, P) \implies (C, P)_{disp} = \vec{0}$
name_null_disp	Transform	Some names imply that the transform should have null displacement	$T_{name} = N \implies T_{disp} = \vec{0}$
sig_null_rot	Transform	Some signatures imply that the transform should have null rotation	$(C, P) \implies (C, P)_{rot} = \vec{0}$
name_null_rot	Transform	Some names imply that the transform should have null rotation	$T_{name} = N \implies T_{rot} = \vec{0}$

TABLE IV
RESULTS (USING z THRESHOLD OF 1.0)

Fault	Number of Faults	Number of Files with Fault	Percentage of Files with Fault
self_transform	4	4	0.05
dup_name	50	43	0.51
dup_sig	32	28	0.33
reversed_name	211	154	1.8
rot_degrees	5	5	0.06
int_suffix	40	25	0.30
ned_transform	3	3	0.04
sensor_null	397	354	4.2
sig_implication	24	16	0.19
name_implication	100	39	0.46
sig_null_disp	78	76	0.90
name_null_disp	33	31	0.37
sig_null_rot	88	78	0.92
name_null_rot	65	61	0.72
Total	**1172**	**811**	**9.6**

runtime, it is not a fault for those transforms to have identical signatures, though our linter will report this as an error, as it does not recognize the macros.

2) Our linter may double report certain errors. For example, if there is a non-null transform named named *sample_name* with the signature (*sample_parent* $\to$ *sample_child*), if both the name and the signature should imply that the transform is null, this will be indicated as two different errors. Therefore the total error counts may appear slightly inflated, though the percentages of files with errors are still valid metrics.
3) The linter does not examine any error patterns which consider the relationships of numerical arguments between transforms at the file scope. As we found during the informal study of transform issues, these arguments are commonly magic constants, and developers commonly make commits to existing files to rectify their orders or signs. However, given the limited number of transforms present in a single file, we found examining patterns strongly dependent on numerical values within a file challenging. We leave this to future work on Launch-Linter.

VI. CONCLUSION AND FUTURE WORK

In this research we determined a set of antipatterns which are mistakenly followed in *roslaunch* configuration files. We developed Launch-Linter to detect the presence of these antipatterns and used this to determine the frequency these errors are made in a public collection of ROS projects.

While we focused on *tf* uses within static transforms, there are much more general ways of using *tf* within source code. We recommend following a similar process to statically determine the presence of errors in other uses of *tf*.

VII. ACKNOWLEDGMENTS

I would like to thank Professor Sebastian Elabum and PhD Student Meriel Stein for their support and guidance in completing this research, as well as their patience during the process. I would also like to thank NSF who funded part of this work through grant #1853374.

REFERENCES

[1] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
[2] T. Foote, "tf: The transform library," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, ser. Open-Source Software workshop, April 2013, pp. 1–6.
[3] D. Engler, D. Y. Chen, S. Hallen, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proc. of the eighteenth ACM symposium on Operating systems principles (SOSP'01)*, Banff Alberta, Canada, Oct. 2001, pp. 57–72.