Mining the Usage Patterns of ROS Primitives

André Santos, Alcino Cunha, and Nuno Macedo
INESC TEC & Universidade do Minho, Braga, Portugal

Rafael Arrais and Filipe Santos
INESC TEC, Porto, Portugal

***Abstract*— The *Robot Operating System* (ROS) is nowadays one of the most popular frameworks for developing robotic applications. To ensure the (much needed) dependability and safety of such applications we forecast an increasing demand for ROS-specific coding standards, static analyzers, and tools alike. Unfortunately, the development of such standards and tools can be hampered by ROS modularity and configurability, namely the substantial number of primitives (and respective variants) that must, in principle, be considered. To quantify the severity of this problem, we have mined a large number of existing ROS packages to understand how its primitives are used in practice, and to determine which combinations of primitives are most popular. This paper presents and discusses the results of this study, and hopefully provides some guidance for future standardization efforts and tool developers.**

I. Introduction

The *Robot Operating System* (ROS) has emerged as one of the most popular frameworks for the development of robotic software, with an explosion of applications and attempts to port it into a fully-fledged industrial framework[1]. Strongly promoting an open-source policy, there are currently over one thousand publicly accessible GitHub ROS repositories, officially indexed in the most recent distribution of ROS[2].

However, this popularity growth has not been accompanied by effective support to promote the dependability of the resulting robots. ROS systems are often developed by a community that is not proficient with standard software engineering practices. Despite attempts to enforce quality metric thresholds[3] and coding styles[4], their adoption has been negligible [1], not only because their benefits are not evident to the ROS developer, but also due to the lack of automated support. Developing a new robot requires the integration of many complex subsystems, such as perception, motion planning, reasoning, navigation, and grasping. Bohren et al. [2] noticed that even with an extensive validation process for each of these individual components, the subsequent step of integrating them into a robust heterogeneous system is a hard task which is not solved yet.

To push the quality of ROS systems forward, stricter coding standards and more advanced analysis tools will undoubtedly be required. ROS-specific *static analysis*, in particular, should be able to analyze not only the behavior of each particular subsystem, but also the integration and interaction of such coexisting components. However, the development of such tools tailored for ROS is far from trivial, due to a few particular challenges. First, ROS allows a high degree of freedom when it comes to *design*, making it difficult to reverse engineer the architecture of the system. In particular, the content of ROS launch files – which are used to effectively deploy a robot – can be fully customizable from environment variables, command-line arguments or configuration files. Second, ROS provides a myriad of *primitives* – used to essentially manage the communication and synchronization between nodes. Such primitives embody different communication paradigms, and can be called freely by a ROS system throughout its lifeline. Finally, there is the inherent complexity of the *languages* of choice for developing ROS systems: C++ , Python, JavaScript and LISP. Consider as an example, the `rosgraph` tool, that constructs the computation graph for a ROS system in runtime. Determining such graph in static time would be extremely useful, but would be exceedingly complex to implement for an arbitrary ROS system, due to the potential complexity of the source code and configuration files. Yet, under a controlled subset of ROS features, that task could be feasible or even straightforward.

In order to more precisely quantify the impact of the first two challenges, we have mined a considerable ROS repository corpus with the goal of detecting common usage patterns of ROS functionalities and primitives, both in configuration files and in the source code. Concerning the third challenge, we have focused on ROS robots implemented in C++ . More specifically, we have analyzed over 400 packages, featuring over 300 launchable applications (the notion of *application* is precisely defined in Section III-B), that are mainly building blocks for larger, more complex robotic systems. This paper presents and discusses the results of this work.

The main outcome of this study is a ranking of the most frequent usage patterns, making it easier for future (static analysis) tool developers to identify where to best invest their effort. In particular, we expect to provide a glimpse of the potential coverage of future static analysis tools depending on which ROS functionalities would be supported. Hopefully, such results will also provide the ROS community with novel insights regarding their software development practices, promoting the development of better programming guidelines.

The paper is structured as follows. Section II presents related work, followed by an overview of ROS in Section III. Section IV describes the methodology employed to collect the usage patterns, while Section V presents and discusses the results of analyzing the collected data. Section VI wraps the paper and points to future work directions.

[1] `https://rosindustrial.org`
[2] `https://github.com/ros/rosdistro/`
[3] `http://wiki.ros.org/code_quality`
[4] `http://wiki.ros.org/CppStyleGuide`

II. Related Work

Research on source code static analysis of robotic systems is scarce, especially on techniques tailored for ROS. In [3] the authors present and explore four static analysis techniques that could be relevant for analyzing robotic software, but concrete solutions on how to adapt them to ROS applications are not proposed. This team has previously proposed HAROS [1], a framework for the static analysis of ROS software[5]. Although it was successful in collecting basic quality metrics, advanced static analysis techniques were quickly encumbered by the complexity of ROS applications. Alternatively, some authors have proposed the (manual) translation of ROS C++ source code into languages more amenable to being statically analyzed, like SPARK [4]. A literature review on safety certification practices [5] has not detected the application of static analysis tools to standard robotic development frameworks, but only the verification of robots implemented in formal specification languages.

As far as we are aware, no studies on mining ROS repositories for typical usage have been published. Techniques for mining software repositories, and in particular for extracting typical API or call usage patterns, have received considerable attention lately [6], borrowing data mining techniques to detect frequent item-sets and sequences [7]. Besides not being tuned for C++ source code, they would probably be overkill for our rather simple queries. Moreover, our study focuses on ROS aspects that are too specific (e.g., launch files) to be analyzed using off-the-shelf techniques. Certain functionalities of HAROS were used to ease this process.

III. Robot Operating System

This section presents the ROS concepts and features targeted by this study. The main organization unit in ROS systems is the notion of *package*, containing several configuration and source code files. Each package is defined by an XML manifest file, specifying, besides meta-data, the building and running dependencies. The official distribution defines a set of packages, as well as their source GitHub repositories (each repository may contain several packages). For the purpose of this paper, the ROS Indigo Igloo distribution was considered. The ROS computation graph is comprised by *nodes* that communicate through *topics* under a publisher-subscriber or client-server paradigm. Communication is provided by a special node, the ROS Master, that also provides access to a shared *parameter server* that acts as a central key-value store. A more detailed description of ROS can be found at the official wiki[6].

A. Feature Diagrams

ROS is extremely flexible and configurable, providing a myriad of functionalities and primitives that can be used in different contexts. This study aims to assess the potential to have ROS systems statically analyzed, so it is relevant to know exactly how the different variants of these primitives are used, for example, whether they are invoked with literal (constant) arguments or otherwise (e.g., a variable containing a value computed in runtime). To aid visualizing this variety, we will characterize ROS applications using *feature models* [8], diagrams initially developed with the goal of modeling alternative configurations in software product lines. Feature models are hierarchical: a child feature may only be selected if the parent is as well. A feature may be *mandatory* (filled circle), forcing its selection with its parent, *optional* (empty circle), or arranged in *or* groups (filled arcs), from which at least a feature must be selected, and *xor* groups (empty arcs), from which exactly one feature must be selected. Every feature model has a *root* feature that is always present in every configuration, and may contain *reference* features (➤) which point to other feature models. Finally, *requires* (⇒) constraints allow the enforcement of cross-tree dependencies. A software product is defined by a selection of features according to these constraints.

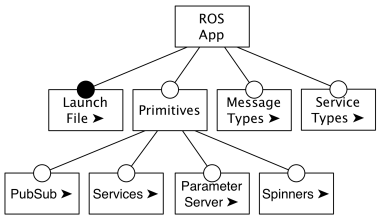

Fig. 1: Application features.

B. ROS Applications

Robotic systems are deployed through the definition of *launch files*, XML configurations used to deploy standalone applications or components for more complex systems. In particular, launch files define which nodes should be launched from the packages, and with which arguments. For the purpose of this paper, we consider a robotic *application* a top-level launch file[7] (ROS App, Figure 1). The launch file/package relationship is many-to-many: launch files may depend on several packages to be deployed, and the same package may be executed by several distinct launch files. By definition, a ROS App contains at least one Launch File, it may contain C++ source files that use some of ROS Primitives, and can declare context-specific Message and Service Types, other than those provided by the ROS core.

Launch files are highly parameterizable (Figure 2), being programmed with Tags. Tags define Nodes to be launched in a given configuration. It is possible to define the host machine (Machine Ref) and whether it is Required (if the node fails, the whole launch fails) or it should Respawn (if the node fails, it will be launched again). Additional Node Arguments can also be passed directly to nodes. Another relevant feature is the notion of Nodelet, a special kind of node designed for high-performance communications by launching multiple nodes in a single process. Topic Remappings can be used to modify the communication between nodes, while Machine

[5] https://github.com/git-afsantos/haros
[6] http://wiki.ros.org
[7] In practice, nodes may be launched directly with the `rosrun` command, but such ad hoc applications are not amenable to be statically analyzed.

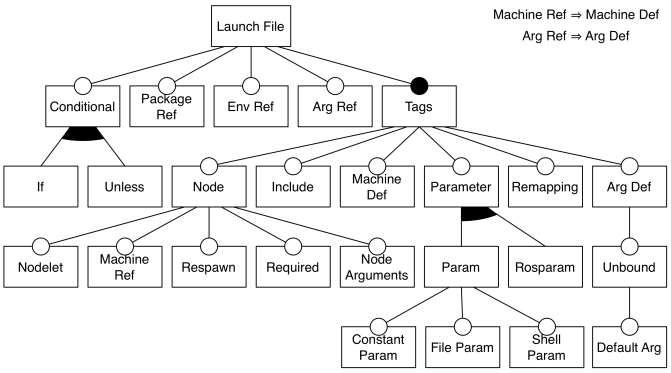

Fig. 2: Launch files.

Def declares different machines on which to deploy nodes. Other launch files can also be Included. Values can be assigned to the Parameter server, that can be later retrieved by the nodes in runtime. Such bindings can be for individual parameters (Param), through various mechanisms, or in bulk through a YAML configuration file (Rosparam).

Local arguments can also be defined (Arg Def). They can be declared with a constant value (which can not be overridden), or left Unbound, in which case a value can be defined as default (Default Arg), which can be overridden by a parent launch file or from the command-line. These arguments can then be freely referenced throughout the launch file (Arg Ref). Environment variables (Env Ref) and paths to other packages (Package Ref) can also be referenced. Finally, every tag can be conditionally executed depending on a variable being true (If) or false (Unless).

C. ROS Primitives

The ROS C++ libraries provide many different overloads for advertising and subscribing topics that allow for some degree of customization. When using the publisher-subscriber paradigm (PubSub, Fig. 3), nodes Advertise topics (which may be Latching, saving previously sent messages, and be notified regarding Subscriber Status) prior to Publishing messages. Other nodes may Subscribe to topics (which may provide Transport Hints to specify the transport layer) by providing a Callback procedure, which can take the shape of a Function, class Method or Functor object. For these primitives, message queue sizes must also be provided (Queue Size, Fig. 5), with size 0 denoting Infinite. Topic names must also be provided for all these primitives (Topic Name, Fig. 5). Message types for these topics may belong to the ROS core (defined in `common_msgs` and `std_msgs`) or be context-specific (Non-std Type). Alternatively, client-server communication is performed through Services (Fig. 3). When nodes Advertise Services, the callback methods to be invoked by Service Clients must be provided. Topic names must also be defined for service primitives. Since we are interested in determining the impact of the usage patterns of these primitives in potential static analysis techniques, we also collect whether they occur Nested in a control structure.

ROS provides primitives to control node spinning (Spinners, Fig. 4). Nodes may wish to work at a given frequency and thus declare a Spin Rate, or have a finer control over the exact amount of sleeping time by declaring a Duration. Regarding the Parameter Server (Fig. 4), primitives allow nodes to Get and Set the value of parameters in runtime. Since one may attempt to get a value not previously set, a default can be provided (Default Param).

The arguments for all these primitives (topics, queue sizes and spinners) may be assigned values through different mechanisms. In this study we are interested in detecting whether these are just Literals or Otherwise computed in runtime (Arguments, Figure 5).

IV. Research Methodology

This section describes the methodology followed in this empirical study, including which kind of information was collected and how the process was operationalized.

A. Research Questions

The main goal of this work is to detect common usage patterns for ROS functionalities. Relevant analyses include the used primitives, their context (e.g., are they in nested control flows?), and with which arguments (e.g., what are the typical queue sizes?). Concretely, this study focuses on answering the following questions.

- RQ1 *What kind of features are typically used in ROS **launch** files to deploy applications?*
- RQ2 *Which, and how frequently, are ROS communication **primitives** are used?*
- RQ3 *In which **context** are these primitives effectively used and how are their **arguments** defined?*
- RQ4 *How is the **parameter server** used in ROS applications?*
- RQ5 *Is there any correlation between **message** and **service** types and other ROS features?*

To answer these questions, we collected, for each analyzed application, which features of the feature models presented in Section III were used. Besides feature usage, additional information concerning concrete values for some of the attributes was also collected, in order to better characterize the applications (see Section IV-C).

B. Repository Selection

Our perspective is for future static analysis tools to be application-centric: a ROS developer would ideally provide a launch file defining the package dependencies and source code of the robotic system that is to be analyzed. Thus, in this study we targeted real ROS robotic systems for which the source code is available online, rather than arbitrary, potentially unrelated packages. We analyzed 13 such systems, listed in Table I, ranging from domestic to field and industrial applications, distributed over various Git repositories and containing hundreds of packages. These systems are highly modular, each providing several ROS applications amounting to different launch configurations.

Some repositories were discarded, amounting to about 100 packages, because their contents were composed mostly by

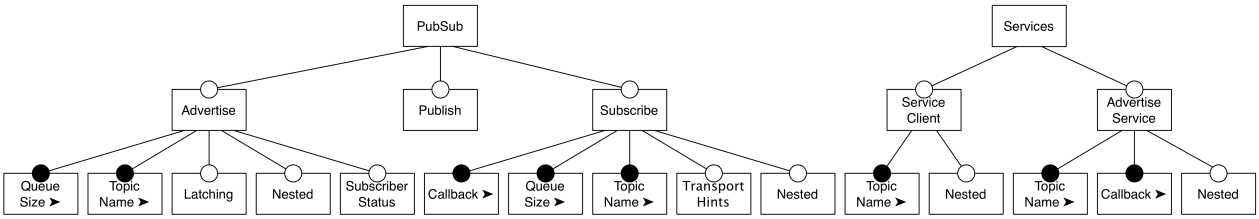

Fig. 3: Communication primitives.

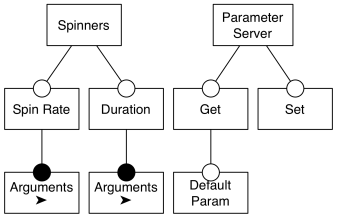

Fig. 4: Spinner & parameter server primitives.

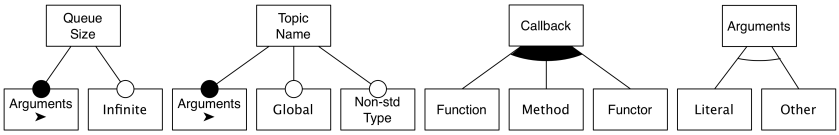

Fig. 5: Argument configuration.

Name	Packages	Apps	C++ LOC
Aubo	11	9	4 773
Fraunhofer IPA Care-O-Bot	97	49	783 120
Clearpath Grizzly	12	15	1 912
Kinova MICO	8	5	4 101
Yaskawa Motoman	10	16	8 376
Robotiq Adaptive Gripper	15	3	3 224
Robotnik AGVS	7	9	2 068
Robotnik GUARDIAN	13	19	5 430
Robotnik RB-1	17	23	502
Robotnik RBCAR	9	11	900
Robotnik SUMMIT	15	7	2 773
Shadow Dexterous Hand	61	41	37 978
Turtlebot	100	136	38 061
Indigo Igloo distribution	319	274	771 994
ROS-Industrial	39	37	27 068
Unique packages	419	343	928 579

TABLE I: Summary of analyzed repositories.

Python scripts, configuration files, 3D models of robots, or applications to enable Android compatibility. While these packages could increase our coverage, they would have no real influence on the results, due to the scarce amounts of C++ code. In the end, we settled with 481 unique packages. From these, 62 were meta-packages that only aggregated other packages (not counted in Table I), and 175 effectively contained C++ code. 207 packages contained at least one launch file, totaling 365 launchable ROS applications (i.e., top-level launch files, as defined in Section III-B); as expected, the package/application relationship is many-to-many, as can be seen in Table I. The table also discriminates how many of these packages and applications belong to the ROS Indigo Igloo distribution or to the ROS Industrial repositories.

C. Tool Overview

The analysis tool is composed of two families of components, implemented as Python scripts, focusing on the analysis of ROS launch files and ROS C++ source code, respectively. The former distinguishes our tool from other generic C++ source mining tools. The choice of Python as the programming language was made out of convenience, since it provides a number of libraries and tools helpful for the intended analysis, such as `Clang`'s Python bindings to extract an Abstract Syntax Tree (AST) from C++ source files, and the `roslaunch` tool, which provided a basis for our launch file analyzer. Furthermore, it allowed to leverage the capabilities of the HAROS framework, by integrating these analyzers as plug-ins that simply implement the analysis logic. The access to all required files and ROS packages is then automated by HAROS.

To mine C++ source code we implemented a module that traverses the AST and converts it to an internal, simplified, model of the language that makes the handling of functions and other language constructs more manageable. This structure is then passed to another module whose responsibility is to traverse the structure and collect the desired statistics regarding communication primitives (RQ2), spinners and parameter server primitives (RQ4). This module registers additional context information, such as the level of control flow nesting of the occurrences and how the parameters are defined (RQ3). Message and service types are also collected and associated with these primitives and their arguments (RQ5). The results are exported in CSV format, to ease inspection and manipulation in a spreadsheet editor.

To analyze launch files (RQ1), we have essentially replicated the functionalities of the widely used `roslaunch` tool, with a twist. Instead of parsing the files and actually launching ROS nodes, our tool stores and processes the information. From this, we are able to gather various simple statistics, including every used tag, as well as the usage of variable references and conditionals. Since launch files can have variability – a consequence of dynamic values and allowing conditional expressions – the analysis includes an interpretation and substitution of the dynamic values. As a result, we are able to determine, e.g., which nodes and topic remappings will effectively have an effect in runtime, as well as arguments left unbound. For those situations, we are also able to identify which variables caused the said alteration. Parameter definitions provide insights regarding RQ4.

Our tool also joins the results from both analyses. For each application (i.e. top-level launch file), we extract its direct and transitive package dependencies. This allows us to aggregate the C++ statistics by application. A final statistic regards the occurrence of message and service types definition files.

D. Threats to Validity

The main threat regards the representativity of the selected packages. The ROS Indigo Igloo distribution features 2106 packages where a source code repository has been declared. Of those, at least 907 have C++ source code (43%), at least 390 Python source code (18.5%), and at least 757 launch files (36%). Our analysis sample features a total of 481 packages. C++ source code, Python source code and launch files are present in 36.5%, 15% and 43% of these packages, respectively. Moreover, 366 out of the 481 packages are indexed in the official distribution. This means that, even though our sample only covers about 15% of the distribution, the ratios of C++ , Python and launch files follow approximately the same pattern, which gives us some confidence that it is representative of typical ROS repositories and applications. It should also be noted that our analysis requires the compilation of the packages, which justifies the judicious selection of repositories.

Our tool mainly distinguishes between literal or otherwise specified arguments for the ROS primitives. This simplistic approach overlooks the fact that some of these non-literal arguments could still be easily resolved through simple static analysis techniques (including the detecting the influence of the parameter server). Nonetheless, the results of the study present the absolute bottom-line for argument resolution.

V. Results

This section presents the results of our study, along with an elementary analysis from the perspective of potential static analysis techniques. While many of these results may be within expectations (for someone already familiar with ROS), it is important to have the necessary data to back up any assumptions. Furthermore, despite being impossible for a tool to completely understand a ROS system (in general, and due to the reasons mentioned before), these data give us an idea of how many existing systems fit within a group where analysis is feasible. Given the amount of collected data, we can only present some relevant results. However, a repository featuring the complete data set is freely available online[8].

A. Package Overview

A first step to extract relevant information out of the collected data is to look at the global (aggregated) values, from a package by package analysis. The most immediate and clear result is that the publisher-subscriber paradigm is more widely used than the client-server paradigm, with 613 occurrences of the respective primitives against 135, respectively. The ROS community guidelines also support this picture.

[8]https://github.com/git-afsantos/ros_data

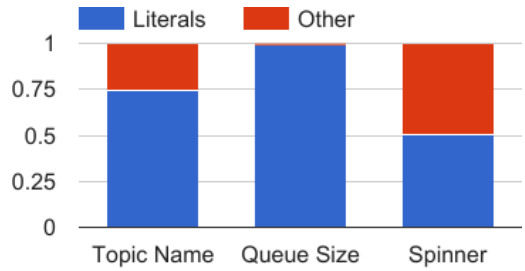

Fig. 6: Usage of literals versus other values in ROS primitives.

Looking at the difference between publishers versus subscribers, or servers versus clients, it is clear that these systems advertise more information than they consume. I.e., there is room to add consumers (planners, etc.) to the analyzed systems. Our data show that the publisher to subscriber ratio is 62%/38%, while for client-server communications we registered 90%/10% in favor of the servers. This discrepancy might arise from a couple of factors: (i) robotic systems are typically designed in pyramidal hierarchical approaches; and (ii) some of the analyzed repositories are meant to be building blocks for more complex robotic systems.

A more interesting analysis, from the perspective of static analysis is to determine how values are passed to the ROS primitives. The analysis scales to new heights whenever arguments are anything other than literals. Fortunately, most examples in ROS tutorials and other resources use literals to define values, and the community tends to follow this pattern as well. Fig. 6 shows the usage ratio of literals versus other methods.

It is clear that over half of all values are declared as literals on the spot. These values reveal no big surprises. ROS discourages the use of global topics (or services) and infinite queue sizes. As expected, their occurrences are rare. Perhaps the most unexpected among these values is the frequency of publisher-subscriber queues of size 1 (54% of all literals). Singleton queues should only be used when there is no interest in processing all messages, but rather only the most recent ones – ROS discards old messages when a queue is full.

Despite the various primitive overloads, our data show that some of these features are seldom used. We registered 88% of all callback functions as being member functions of a C++ class, as opposed to other variants, and no subscriber specifies which transport protocol is preferred (TCP is the default). On the publisher side, only 5% latch messages, and only 2% are notified when new subscribers join. We found that primitives do not occur within loops or conditionals for the majority of registered occurrences, and, as such, the current picture is beneficial from an analysis point of view.

The usage and definition of custom message types may affect the static analysis, since tools might rely on knowledge regarding standard message-types to ease analyses. We determined that 15% of the analyzed packages define custom message types, and that 32% of all publishers and subscribers exchange custom messages.

The last note of this global overview regards the ROS

parameter server. Its documentation states that it is not designed for high performance, making it better suited for shared storage of static values, or values that rarely need to be changed. The data support this, given that only 38 parameters are set during runtime, compared to 705 readings. Out of these readings, 83% declare a default value.

B. Launch File Overview

In our sample we registered 365 launch files deploying a total of 1418 nodes. Out of these, there are only 275 unique nodes (19.4%), confirming that, indeed, many launch files are only variations of specific applications and scenarios. Nodelets are seldom used, amounting to 119 out of the 1418 nodes. Other node-related features are also uncommon. Only 46 nodes are marked as required, and a mere 16 are launched on machines other than the one executing the launch. The only relatively common feature is to mark a node as to respawn, where 39.5% of all nodes are marked so.

Regarding the parameter server, on average, each launch file defines more than 10 parameters. From a static analysis viewpoint, parameters, by themselves, would not be too challenging to inspect. Problems start to arise when said parameters are defined not by static values, but from reading configuration files or by capturing the output of an arbitrary shell command – both allowed by ROS. Out of 3735 parameter definitions, 26% come from configuration files, and 6.4% come from shell commands.

Other aspects worth mentioning are the number of remappings (1009), the number of environment variables read (652), and the number of conditionals (1179). On average, each launch file remaps between two and three names (topics or parameters), reads about two environment variables, and declares over three entities conditionally. These numbers may not be alarming, but for an analysis tool to be complete, additional steps are required. For remappings, the tool must be able to resolve and correctly redirect ROS names and namespaces. For environment variables, user input is required. Finally, for conditionals, the tool has to be able to resolve values, which may come, e.g., from environment variables.

C. Combined Feature Usage

While feature-wise statistics are interesting, from a static analysis perspective it is more relevant to consider the combined usage of different features, namely, what would be the expected coverage of a static analysis tool if only a given set of features is supported.

Two problematic usages of the primitives to consider when developing a static analyser are nested occurrences and non-literal arguments. The latter is even more problematic in ROS due to the potential usage of the parameter server: how to predict in static time the effect of a primitive that is invoked with a value fetched from the server? Our study shows that 69% of the 175 packages containing C++ code and 24% of the applications do not use these features at all, and thus could be handled by a relatively straightforward static analyser. The coverage discrepancy is due to the fact that applications, being composed by many packages, typically end up using at least one package that relies on one of these features.

Unfortunately, our study shows that the gains of additionally supporting only one of these features, that is, either non-literal arguments or nested occurrences, would hardly be noticeable: the former would improve coverage to (80%/30%), and the latter to (71%/31%). Moreover, the parameter server is abundantly used by packages (89%), which means that most likely its effect had to be considered also. Essentially this means that in order to achieve a high coverage of applications it would be necessary to consider the full set of ROS features.

VI. Conclusion

This paper presents some preliminary results on the usage patterns of ROS primitives, thus shedding some light on how ROS software is being developed and used in practice. This understanding could prove useful to the community in various ways. For instance, it could lead to updates in the current programming style guidelines and tutorials in order to clarify less used (or misused) features. It can also help future (static analysis) tool developers to identify where to best invest their effort, namely which features to support in order to maximize their coverage.

Regarding the latter, our main conclusion is that a straightforward static analysis tool for ROS would be able to address a considerable amount of individual packages, but unfortunately a relatively small amount of applications. To increase this coverage, sophisticated static analysis techniques will need to be developed. Another option would be for the ROS community to issue more strict ROS coding guides, namely discouraging the usage of the problematic usage patterns.

In the near future we intend to perform more sophisticated analysis over the collected data, in order to detect other interesting usage patterns and correlations between them. We also intend to add more repositories and applications to our data base, in order to validate the results found so far.

References

[1] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *IROS*. IEEE, 2016, pp. 4491–4496.

[2] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *ICRA*. IEEE, 2011, pp. 5568–5575.

[3] A. Cortesi, P. Ferrara, and N. Chaki, "Static analysis techniques for robotics software verification," in *ISR'13*. IEEE, 2013, pp. 1–6.

[4] P. Trojanek and K. Eder, "Verification and testing of mobile robot navigation algorithms: A case study in SPARK," in *IROS*. IEEE, 2014, pp. 1489–1494.

[5] J. Ingibergsson, U. Schultz, and M. Kuhrmann, "On the use of safety certification practices in autonomous field robot software development: A systematic mapping study," in *PROFES'15*, ser. LNCS, vol. 9459. Springer, 2015.

[6] S. Khatoon, G. Li, and A. Mahmood, "Comparison and evaluation of source code mining tools and techniques: A qualitative approach," *Intell. Data Anal.*, vol. 17, no. 3, pp. 459–484, 2013.

[7] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "Comparing approaches to mining source code for call-usage patterns," in *MSR*. IEEE Computer Society, 2007, p. 20.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.