# 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 Neves dos 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]. ROS encourages an open-source policy, and 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 in 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.

The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 723658. This work is also financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013».

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

To push the quality of ROS systems forward, stricter coding standards and more advanced analysis tools will undoubtedly be required. *Static analysis* is one of the techniques that could benefit this cause, given its suitability to find subtle errors that are difficult to reproduce at runtime. 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 become feasible.

In order to more precisely quantify the impact of the first two challenges, we have mined a considerable ROS repository corpus of over 400 packages and 300 launchable applications (see Section III-B for a definition of *application*), 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++. The analyzed components are mainly building blocks for 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 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.

## 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 patterns 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 a shared and central key-value *parameter server*. A more detailed description of ROS can be found at the official wiki[6].

[5]`https://github.com/git-afsantos/haros`
[6]`http://wiki.ros.org`

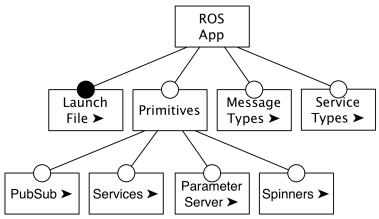

Fig. 1: Application features.

### A. Feature Diagrams

ROS provides a myriad of flexible and configurable functionalities and primitives, usable in various 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., values 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.

### 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, Fig. 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 ROS Primitives, and can declare custom, context-specific Message and Service Types.

Launch files are highly parameterizable (Fig. 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

[7]In practice, nodes may be launched directly with the `rosrun` command, but such ad hoc applications are not amenable to be statically analyzed.

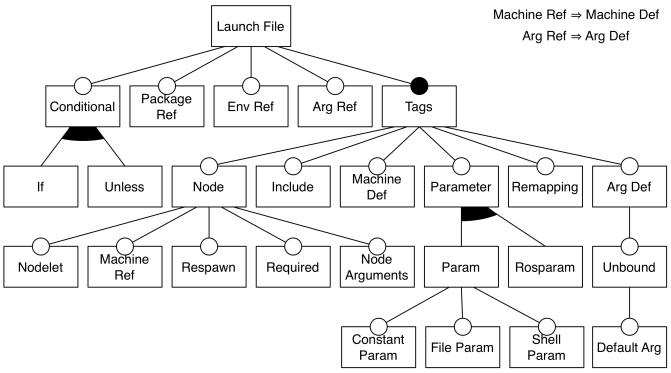


Fig. 2: Launch files.

Node Arguments can also be passed directly to nodes. Another relevant feature is the Nodelet, a special kind of node designed for high-performance intra-process communications. Topic Remappings can be used to modify the communication between nodes, while Machine 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 and freely referenced throughout the launch file (Arg Def and Arg Ref). 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), and be overridden by a parent launch file or from the command-line. 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 (and optionally specify the transport layer with Transport Hints) by providing a Callback procedure that takes 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 (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 loop frequency (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. We are interested in determining whether these are just Literals or any Other kind (Arguments, Fig. 5).

## IV. Research Methodology

This section describes the methodology followed in this empirical study, including what kind of information we 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. Concretely, this study focuses on answering the following questions.

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

From a code quality and analysis standpoint, the answers to these questions dictate how "knowledgeable" a person or tool must be. Knowing which primitives are most used can help prioritize their support in tools. For static analysis, it is also very relevant to know in which context such primitives appear (namely, whether they are within control flow) and how are they parameterized. Analyses scale to new heights when arguments are anything other than literals, and more so when ROS parameters are involved. How to predict in static time the effect of a primitive that is invoked with a value fetched from the server? Does a parameter hold the value defined in a launch file, or has it been redefined in runtime? Finally, the usage and definition of non-standard message types limits the domain-specific knowledge tools can leverage.

To answer these questions, for each analyzed application, we collected 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).

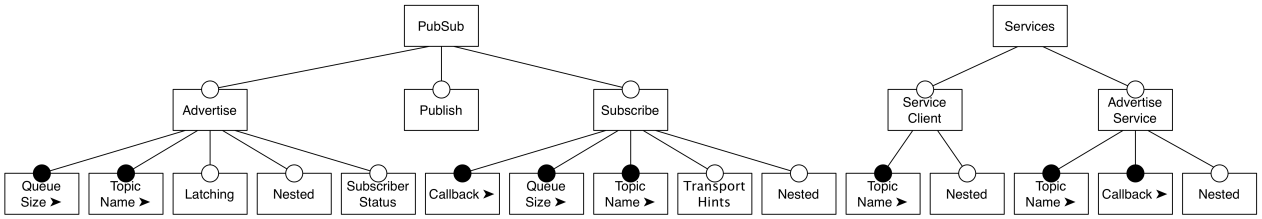


Fig. 3: Communication primitives.

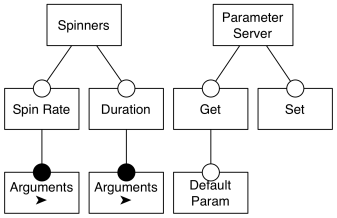


Fig. 4: Spinner & parameter server primitives.

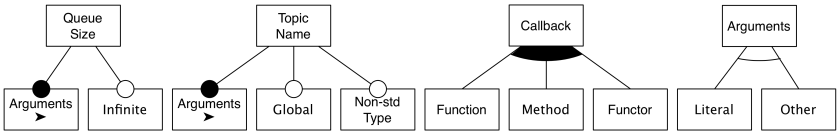


Fig. 5: Argument configuration.

### 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 to be analyzed. Thus, in this study we targeted complete 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 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 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 launch files, 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. Table I also discriminates how many of these components belong to the ROS Indigo Igloo official distribution or the ROS Industrial repositories.

| 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 and apps | 419 | 343 | 928 579 |

TABLE I: Summary of analyzed 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 integration with the HAROS framework in the form of analysis plug-ins, letting HAROS automate the fetching of files and packages.

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 (RQ1), 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 (RQ2). 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 (RQ3), 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 be in effect during runtime, as well as arguments left unbound. For those situations, we are also able to identify which variables caused 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, for RQ5, regards the occurrence of message and service types definition files.

### D. Threats to Validity

The main threat is 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. On the other hand, we acknowledge that nearly half of the packages come from just two development groups, a fact that may skew the results towards the practices they adopted.

## 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 analyze an arbitrary ROS system (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

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

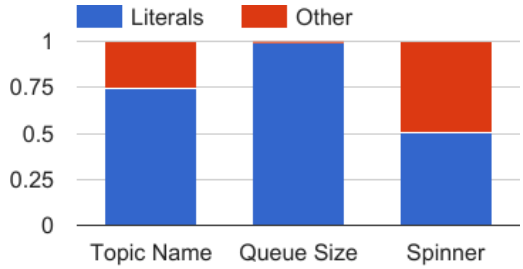

Fig. 6: Usage of literals versus other values in ROS primitives.

clear result, addressing RQ1, is that the publisher-subscriber paradigm is more widely used than the client-server paradigm, with 613 occurrences of the former primitives against 135 of the latter. The community guidelines also support this picture.

Looking at the difference between publishers versus subscribers, or servers versus clients (still on RQ1), it is clear that these systems advertise more information than they consume. 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 the fact that robotic systems are typically designed in pyramidal hierarchical approaches, and some of the analyzed repositories are meant to be generic building blocks for more complex robotic systems.

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 the preferred transport protocol (defaults to TCP). Only 5% of publishers latch messages, and a mere 2% are notified when new subscribers join.

Moving to RQ2, it is evident from our data that over half of all values passed to primitives are declared as literals on the spot. These values reveal no big surprises, and fortunately make analysis easier. Most examples in tutorials and other resources use literals to define values, and the community tends to follow this pattern. Fig. 6 shows the usage ratio of literals versus other methods. Additionally, we found that primitives do not occur within loops or conditionals for the majority of registered occurrences.

ROS discourages the use of global topic names and infinite queue sizes and, as expected, they rarely occur. Perhaps the most unexpected among the inspected 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.

Regarding the exchanged messages, and to answer RQ5, only 15% of the analyzed packages define custom message types. However, 32% of all publishers and subscribers make use of such messages, making this kind of analysis a feature of significance.

The last note of this global overview, in part answering RQ4, 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

Launch files feature a number of things worth considering for RQ3. 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 just 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 specific machines (defaulting to *localhost*). The only relatively common feature, affecting 39.5% of all nodes, is to make a node be able to respawn.

Regarding the parameter server, and concerning RQ4 as well, on average, each launch file defines more than 10 parameters. And, while parameters, by themselves, would not be too challenging to inspect, problems start to arise when their values are defined not by static values, but from 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 about RQ3 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 they require additional analysis steps. For remappings, a tool must be able to resolve and correctly redirect ROS names and namespaces. For environment variables, user input is required. Finally, for conditionals, a tool has to be able to resolve arbitrary 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 prominent and problematic usages of primitives, concerning RQ2, are conditional occurrences and non-literal arguments, especially for arguments served from ROS parameters. 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 supporting just 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 in 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 a comprehensive 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. Adding more repositories and applications to our database, to validate the results found so far, and thus increase the online data set, is a priority as well. Finally, we are aware that our analysis should extend to the remaining ROS primitives, such as *Actions* and the *Dynamic Reconfigure* service. We expect their inclusion to be relatively straightforward.

## 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*. 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*, ser. LNCS, vol. 9459. Springer, 2015, pp. 335–352.

[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, 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.