Static-time Extraction and Analysis of the ROS Computation Graph

André Santos, Alcino Cunha and Nuno Macedo
High-Assurance Software Laboratory (HASLab)
INESC TEC & University of Minho
Braga, Portugal
Email: {andre.f.santos, alcino.cunha, nuno.m.macedo}@inesctec.pt

***Abstract*—The *Robot Operating System* (ROS) is one of the most popular open source robotic frameworks, and has contributed significantly to the fast development of robotics. Even though ROS provides many ready-made components, a robotic system is inherently complex, in particular regarding the architecture and orchestration of such components. Availability and analysis of a system's architecture at compile time is fundamental to ease comprehension and development of higher-quality software. However, ROS developers have to overcome this complexity relying mostly on testing and runtime visualisers. This work aims to enhance static-time support by proposing, firstly, a metamodel to describe the software architecture of ROS systems (the *ROS Computation Graph*) and, secondly, model extraction and visualisation tools for such architectural models. The provided tools allow users to specify custom-made queries over these models, enabling the static verification of relevant properties that had to be (manually) checked at runtime before.**

***Keywords*-static analysis; software architecture; software quality; robotics**

I. Introduction

Robotics is going through a phase of growth and rapid development, with modern robots being applied in industry and research, but also at the consumer level. Human-Robot interaction and free motion in unstructured environments are now a reality that renders traditional, hardware-based safety mechanisms (such as enclosing cages) infeasible, and increases the chances for catastrophic failure. Much of the control over a robotic system is handled by increasingly complex networks of software components. Thus, ensuring the correctness of both the software components and their orchestration, as well as ensuring the availability of analysis tools during development are issues of increasing importance.

The *Robot Operating System* (ROS) [1] is an open source development framework that has thrived and brought together a vast community, mostly due to its ease of access and usage. Many modern robotic applications use ROS either in production or during prototyping stages. ROS provides ready-made implementations for many common features and algorithms, and is highly based on independent processes communicating by message-passing. These factors make the work of a ROS roboticist more akin to that of a system architect or designer, by determining which components to reuse, which to implement, and how they interact.

In ROS, orchestration includes (among other tasks) guaranteeing that all communication channels are correctly assigned on both ends, and that the published messages match the message types expected by subscribers. Since components present themselves at varied stages of maturity and often lacking adequate documentation, without proper tool support this process is cumbersome and error-prone. The goal of this work is to enable ROS developers to reason about the architecture and context-specific properties of a robotic system, assembled with third-party packages, at a level of abstraction that is more intuitive than raw source code.

Tools like `rqt_graph`[1], a popular ROS tool for runtime visualisation of the *ROS Computation Graph* – the network of processes and communication channels – are essential to make the configuration aspects of ROS less error-prone and to improve the comprehension of a system's architecture. But what runtime analysis gains in accuracy, it loses in being more time-consuming, requiring configuration and execution of the system, covering a limited range of execution paths and being harder to trace back to the source code.

Recovering the ROS Computation Graph at compile time would be ideal, as it would allow the architecture to be immediately available to the developer, without interrupting the flow of development. Moreover, we believe that the ability to perform advanced analyses over architectural models is key to promote comprehension and to detect potential issues. Despite the existence of general patterns and properties, what constitutes a good system design is largely context-dependent, thus such analyses must necessarily be customisable by the user. To our knowledge, there are no tools providing this kind of development support as of yet.

This work proposes a solution for the analysis and validation of the architectural design of ROS applications. Our first contribution is a metamodel to describe the software side of ROS systems, including the ROS Computation Graph. This metamodel is close to the level of abstraction adopted in the ROS documentation while still being traceable back to the source code artefacts. This has some immediate advantages: i) familiarity with the concepts; ii) being more tractable to reason about than raw source code; iii) being amenable to reverse engineering from the source code. A

[1] http://wiki.ros.org/rqt_graph

second contribution consists of components for the static-time extraction, visualisation and analysis of models conforming to this metamodel, fully integrated in the core capabilities of HAROS[2], a ROS static analysis framework. HAROS was extended to support architectural analysis plug-ins, and its analysis routine allows the execution of user-defined queries on extracted models. These analyses are natively supported by the visualiser, providing graphical feedback for query matches. Since extracting a fully-resolved model at static time may be impossible, the metamodel and the components natively handle incomplete or ambiguous information. It is important to note that, although ROS version 2 has been officially released, our work targets the more widely adopted original version of ROS.

This paper presents our proposal starting with an overview of the most relevant features of ROS and how they fit into our metamodel in Section II. Right after, Section III presents the tools and methodology employed to extract, visualise and query ROS architecture models. Section IV reports our experience with a real case study, and illustrates some of the properties that can be checked with our query system. In Section V we compare our approach to existing work in ROS modelling and static analysis. At last, Section VI wraps the paper and points to future work directions.

II. A Metamodel for ROS Applications

ROS systems are complex to describe, much due to their flexible and informal nature. Nonetheless, some key concepts and features are present in most situations. Our metamodel and analyses try to capture these. For most of the entities in the metamodel a *Location* (in the source code, with varying granularity) and *Source Conditions* are provided. Conditions can refer to anything that might interfere with the manifestation of the entity, and whose value is unknown after the extraction takes place, for instance the conditions within an `if` statement. Conditions allow us to consider conditional entities and variability points as first-class items.

Our catalogue of concepts can be roughly divided in two groups, one for *Source Code Entities* – entities that are static and whose purpose is to build the ROS system – and another group for *Runtime Entities* – entities that exist and make sense only during the runtime of a specific ROS system. In Fig. 1 we provide the source code for a minimal ROS application that will be used as a running example in this paper.

Source Code Entities are the most familiar to any developer, and the most prominent of them is the *Package*, which is the core build and distribution unit in ROS. A Package, uniquely named within a system, consists of a set of configuration and source code files, plus any additional required resources, such as data files. In particular, a Package may contain *Launch Files*, XML configuration files that are used to define and deploy Runtime Entities (Fig. 1 line 27 onward). We abstract most of these contents as simply *Source Files*. Packages are often grouped and published in *Repositories*, with GitHub being the most popular hosting platform.

[2]https://github.com/git-afsantos/haros

```
1  // File "active.cpp", compiles to Node "active"
2  int main(int argc, char **argv) {
3    ros::init(argc, argv, "talker");
4    ros::NodeHandle nh; std::string param; double hz = 10;
5    std::getline(std::cin, param);       // <-- user input
6    nh.getParam(param, hz);  // <-- impossible to extract
7    ros::Publisher pub =
8        nh.advertise<std_msgs::String>("monologue", 10);
9    ros::ServiceClient client =
10       nh.serviceClient<example::GetCounter>("counter");
11   while (ros::ok())
12     { /* pub.publish(...); client.call(...); */ }
13   return 0;
14 }

15 // File "reactive.cpp", compiles to Node "reactive"
16 void callback(const std_msgs::String::ConstPtr& msg);
17 bool getCounter(example::GetCounter::Request &req,
18                 example::GetCounter::Response &res);
19 int main(int argc, char **argv) {
20   ros::init(argc, argv, "listener");
21   ros::NodeHandle nh;
22   nh.subscribe("chatter", 10, callback);
23   nh.advertiseService("counter", getCounter);
24   ros::spin();
25   return 0;
26 }

27 <launch>    <!------- File "minimal.launch" ------->
28   <param name="frequency" type="double" value="1"/>
29   <node name="listener" pkg="example" type="reactive"/>
30   <node name="talker" pkg="example" type="active">
31     <remap from="monologue" to="chatter"/>
32   </node>
33 </launch>
```

Fig. 1. A minimal ROS application, featuring two C++ nodes and an XML *launch file* to configure the application deployment.

Packages are extended with the set of *Nodes* they contain. The term *Node* is often used in ROS both to describe a runtime process and the executable from which processes are instantiated (also called the *type* of the runtime node). To avoid ambiguity, we refer to a Node as a compiled program, script or some other build target, while *Node Instances* denote the runtime counterparts. As an example, compiling *active.cpp* and *reactive.cpp* from Fig. 1 would result in two Nodes. Note that a single Node can be instantiated multiple times within the same application, with different arguments.

Each Node contains the set of Source Files that builds it and a set of *ROS Primitive Calls*, function calls to ROS primitives in the source code, such as *advertise* (line 8) or *subscribe* (line 22). Each Primitive Call stores all relevant function arguments.

Runtime Entities consist of a series of concepts to which any ROS developer is exposed early on. Most of these were directly identified from the official ROS documentation[3][4]. At top-level there is the *ROS Computation Graph*, the network of named entities processing and sharing data in a ROS system. These entities, also known as *Resources*, can be divided into four main categories, namely *Node Instances*, *Topics*, *Services* and *Parameters*. Even though there are extensions to these Resources added by libraries, such as *Actions*, these are not yet included in our metamodel.

Node Instances are the main participants in a ROS system, since they are responsible for all data processing. The remaining

[3]http://wiki.ros.org/ROS/Concepts
[4]http://wiki.ros.org/Names

three types of Resources represent the main available ROS primitives through which Node Instances communicate. In essence, Topics route messages in a publisher-subscriber paradigm, Services are the client-server (or remote procedure call) alternative, and Parameters are shared key-value pairs of data, stored in a central Parameter Server.

All Resources are uniquely identified by a *ROS Name*, a pseudo-identifier following some naming and resolution conventions (cf. the documentation). The Node Instances in our example are given the names *listener* and *talker* (see lines 29 and 30). Node Instances gain access to other Resources by supplying the respective ROS Names to the different primitives (e.g., *chatter* in line 22). Also, as a way to promote component reuse, it is possible to provide Node Instances with *Name Remappings* (line 31). Remappings transparently re-route the names supplied to primitives. This is useful to connect or disconnect nodes without altering the source code.

It is possible for the model not to have enough information to fully resolve ROS Names. In this case, as many parts as possible of the ROS Name are resolved (a prefix or suffix) and then wildcards are used to capture the rest. The *frequency* parameter in our example (defined at line 28) is accessed by providing the Parameter name via user input (lines 5 and 6). As such, it is unknown at static time, and the Parameter name will be referenced in the extracted model with a wildcard *?*.

Our metamodel also considers *ROS Primitive Links*, which are the connections between Resources, or, in essence, the edges of the Computation Graph. They include an additional reference to the resolved ROS Name that was initially requested in the corresponding Primitive Call, before applying any name remappings of the specific Node Instance. Consider the call to *advertise* in the *talker* Node Instance from our example (line 8). This call is affected by the remapping from *monologue* to *chatter* in the launch file (line 31). As such, a *Publish Link* between *talker* and the Topic *chatter* is derived, but it also retains the initial reference to *monologue*.

Connecting both Source Code and Runtime Entities, we introduce the concept of *Configuration*. Configurations are user-defined ROS applications, composed of the corresponding Computation Graph, sequence of Launch Files, and required environment variables. Including a copy of the environment is a necessary step, since environment variables can be accessed from all sorts of source files, and even be part of Conditions.

Lastly, we introduce the entry point to the whole metamodel, the *Project*, which is a user-defined set of Packages and Configurations, and may also contain a set of relevant Repositories. Fig. 2 summarises the presented concepts.

III. Architectural Analysis in HAROS

Components for the extraction and analysis of a ROS system's model after-the-fact from source code have been developed and integrated as core parts of the HAROS framework. An *extractor* populates a Computation Graph database, which is consumed by a graphic *visualiser* and a general query *analyser*, as detailed in the succeeding sub-sections.

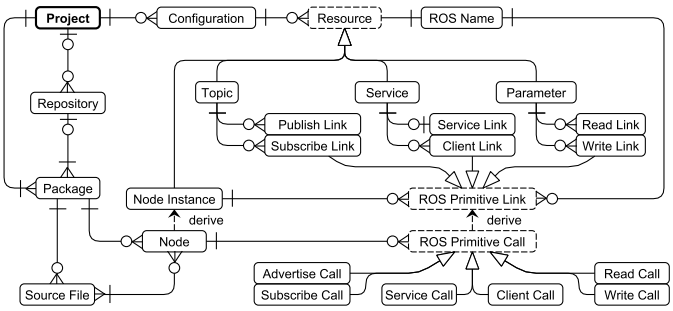

Fig. 2. ROS metamodel. *Location* and *Source Condition* are omitted. Edges use the Entity-Relationship Diagram notation for cardinality.

```
1 project: running_example
2 packages: [example_package]
3 configurations:
4     minimal:
5         launch: [example_package/launch/minimal.launch]
```

Fig. 3. A HAROS project file declaring a *minimal* Configuration.

To move towards application-centred (rather than package-centred) feedback, HAROS project files were extended to support the definition of Configurations (an identifier and a list of launch files, see Fig. 3). Since the extraction process can be incomplete, users are also able to provide *extraction hints*. These hints consist of listing Node Instances annotated with the Topic or Service names and message types that they should advertise or subscribe. Extraction hints may be partial, and they are used to find potential matches when Resource names are unresolved in the normal extraction procedure.

Technically, these components required new additions to HAROS' core, including a CMake parser, a Launch file parser, a Node extractor and a Configuration extractor. Working directly with a compiler's *Abstract Syntax Tree* (AST) is still rather complex and hardly extensible (e.g. the variants of a `for` loop have distinct trees), so an intermediate step is performed to convert the C++ compiler's tree into a simplified data structure. Given that this is not a ROS-specific process, this component has been refactored and distributed as a standalone library[5]. Despite its current focus on C++, the library has been designed for extensibility, so that Python, C and other relevant languages in the ROS ecosystem might be integrated at a later point.

A. Model Extraction

Model extraction relies on the HAROS infrastructure to retrieve ROS source code and generate the corresponding Computation Graph, according to our metamodel.

The first step of this reverse engineering process is to identify the application under analysis, but ROS has no clear definition for *application*. In fact, ROS is dynamic, allowing participants (nodes) to join or leave the system at any time. Our first approach [2] attempted to define a ROS application as being a top-level launch file within a package – a launch file that is not reused by any other. This approach enables the

[5]https://github.com/git-afsantos/bonsai

automatic detection of applications and is accurate to some extent. However, it is too restrictive, since several top-level launch files may be launched in conjunction, forming a single application. Thus, in this work, defining which launch files and nodes go into a Configuration is a task delegated to the user.

The next step is to parse the launch files in order. The parsing process is very akin to a live interpretation of the file, similar to what the `roslaunch` tool does, resolving values as it progresses. In this case, however, we are only interested in registering launch artefacts, rather than actually launching them. From the launch file in Fig. 1, this step would register the creation of a Parameter and two Node Instances, one of which contains a remapping. Our parser also differentiates itself from `roslaunch` by not throwing errors or resorting to default values when it cannot resolve variables. Instead, it marks the affected entities as *conditional*, registering the Conditions that it is unable to resolve. This information can be used to determine how coupled a system is to external variables. The outcome of this parsing step is a fully resolved set of Node Instances, Parameters, remappings and other artefacts that are (definitely or conditionally) part of the Configuration.

The main challenge with node declarations in launch files is that nodes are instantiated by providing package and executable names (*pkg* and *type* attributes in Fig. 1, lines 29 and 30). Packages and Source Files can be extracted using the infrastructure of HAROS. To build the mapping between Source Files and Nodes, we have to parse the various CMake files used by the build system.

The final step to the extraction process is the parsing of the source code itself into ASTs. By traversing the AST of a Node, it is possible to detect occurrences of ROS Primitive Calls and whether they are under control flow structures. In the event that the Primitive Calls contain non-literal arguments, the extractor attempts to walk backwards in the program to resolve the unknown values. An example of this could be the *param* variable in line 6 of Fig. 1. In this case, however, the expression would be impossible to resolve.

Parsing and traversing an AST can take some time, especially when parsing C++ code. To improve the overall performance of the extraction process, we started working on a built-in database of well-known, pre-parsed Nodes. This database is a work in progress, but it should only contain Nodes that belong to documented packages, released in the official ROS distributions. One of the major principles in ROS is the reuse of components. Considering that third-party packages are often installed from binaries, and source code may not be available for parsing, embedding this domain knowledge in the tools contributes to the completeness of the extracted models.

After traversing and extracting primitives from all Nodes, it is possible to match Nodes with launch files (e.g. matching line 30 from the example with *active.cpp*), derive the respective Node Instances and ROS Primitive Links, and apply the given extraction hints. It is through the instantiated Links that we determine which Topics, Services and Parameters need to be created and added to the Configuration (e.g. the Topic *chatter*).

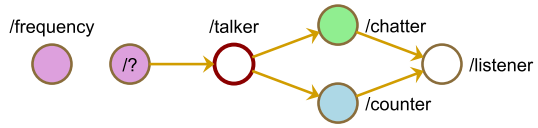

Fig. 4. Rendering of the toy example from Fig. 1. Resource types are distinguished by colour. The highlight on *talker* denotes a query match.

```
1 for n in </configs/nodes> where len(n.remaps) > 0 return n
```

Fig. 5. Basic query to identify Node Instances with remappings.

B. Model Visualisation

The Computation Graph visualisation component, integrated in the HAROS visualiser, renders a graph of the extracted model, where each graph node is a Resource and the edges represent the ROS Primitive Links between them. This component features individual Resource inspection (e.g. message type and traceability to the source) and visibility settings for different Resources. Fig. 4 shows the rendering of our toy example.

The visualiser has also been built to natively support the incomplete nature of extraction processes, both by displaying the conditions associated to a conditional Resource and by rendering conditional entities differently, using dashed lines. Moreover, when a Resource name is unresolved, the visualiser computes and highlights every potential match of the same type to help the user identify possible alternative configurations. For instance, selecting the *?* Parameter from Fig. 4 would highlight *frequency*, which is most likely the correct match.

C. Model Analysis

The analysis procedure of HAROS has been altered in two ways, so as to enable both fine-grained and coarse-grained analysis of Computation Graphs. The former can be achieved by developing custom analysis plug-ins, as is the norm with HAROS. In addition to the original entry points for source files and packages, a new plug-in entry point is available for Configurations produced by the extractor. This gives programmatic freedom to the plug-in developer over the Configuration data structure. Coarse-grained analysis, on the other hand, trades flexibility for ease of use. We have extended HAROS with an analysis component that runs user-defined queries over the extracted models. In many cases, using a query language is simpler, and more desirable, than implementing an analysis plug-in, but the range of expressible properties is more limited. As an example, Fig. 5 shows a simple query to identify Node Instances affected by name remapping.

Such a query engine could be implemented as a HAROS plug-in itself. Query matches are reported back to the user through the same issue reporting mechanisms that are available to traditional plug-ins, but, due to embedding the engine as a core feature of HAROS, the Computation Graph visualiser can also provide graphical feedback for applicable queries. Fig. 4 shows the highlight on *talker*, which is the match found for the query in Fig. 5. Another immediate advantage of embedding

the engine is that queries can be specified in the project files that HAROS already requires.

The query language uses a Python-like syntax and the implementation of the query engine is based on *PyFlwor*[6], a query system for in-memory Python objects. PyFlwor allows for two styles of queries, one based on XPath (called *path expressions*) and another based on XQuery (called *FLWR* expressions). Both styles of expressions operate on collections, and return sets of objects. Path expressions have a declarative flavour, whereas FLWR expressions are more programmatic. The FLWR acronym stands for *For-Let-Where-Return*, the four basic statements of that style, in order. The example given in Fig. 5 follows the FLWR style, without a *let* statement.

A path expression can be refined with a filter (called *where condition*) and combined with various set operators (e.g. union, intersection, difference). The query in Fig. 5 can be rewritten as the following path expression: `configs/nodes[len(self.remaps) > 0]`. The *where* condition is the expression within square brackets.

A FLWR expression starts with a *for* statement, iterating over a set of elements given by a path expression. An optional *let* statement follows, where additional variables can be defined. The value of a variable can be the result of a nested FLWR query. The third statement, an optional *where* statement, is used to define arbitrary conditions. When a condition is satisfied, the *return* statement is executed, to calculate the desired result for that iteration and add it to the set of values that is to be returned at the end. Section IV-B contains more complex examples following this style.

To recapitulate, our contribution is not the query language or the engine themselves, but integrating this engine in HAROS and providing the context on which the engine operates, i.e. making certain functions and collections of objects available to users of the query language (e.g. the collection of all parsed Nodes). Namely, queries have access to four top-level collections: *files*, the collection of all Source Files; *packages*, the collection of all Packages; *nodes*, the collection of all extracted Nodes (not Node Instances); and *configs*, the collection of all extracted Configurations. Each of these provides all the attributes and relations defined in our metamodel.

IV. Preliminary Experiments

In order to validate our approach, we have tested our extraction tools on real case studies, such as the TurtleBot2[7], and the AgRob V16[8]. The former is a robot that is commonly used in the ROS community for learning and research. The latter is a modular robotic platform for hillside agriculture, adapted to sloping and uneven terrains (see Fig. 6). It is designed for monitoring, precision spraying, pruning and selective harvesting, particularly in steep slope vineyards. In this section we present the extracted models for both case studies and our motivations for selecting them, as well as some queries performed over the models.

[6]https://github.com/timtadh/pyflwor
[7]http://www.turtlebot.com/turtlebot2/
[8]http://agrob.inesctec.pt/

Fig. 6. The AgRob V16 agriculture robot monitoring a slope vineyard.

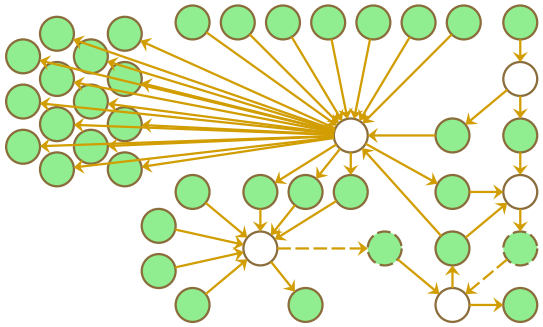

Fig. 7. Rendering of a TurtleBot2 Configuration, including Node Instances (white), Topics (coloured) and conditional Topics (dashed). Names and Parameters are omitted for readability.

A. Model Extraction for TurtleBot2 and AgRob V16

TurtleBot2 provides for an interesting case study, since its source code includes uncommon features, such as conditional topic subscriptions, that our tools should be able to handle. Besides, it has been implemented with extensibility and customisation in mind, featuring tens of different launch files and configurations out-of-the-box. In fact, this example was enough to demonstrate the limitations of the C++ parser. Some of the required arguments for ROS primitives are provided by member variables of a C++ class – internal state that may be modified at any other point of the program – and values that are constructed within loops (e.g. subscribing topics from a Parameter list). Fig. 7 shows an extracted model that required seven topic hints to resolve all wildcards. This figure also shows a large number of disconnected topics, which confirms the design for extensibility.

The main issue with TurtleBot2 is that it is an academic example. AgRob V16, on the other hand, is closer to the average industrial robot. First of all, it is a more complex system, as is evident from the extracted model in Fig. 8a. Secondly, most of its software comes from integrated third-party packages that provide mapping and localisation, among other features, while TurtleBot2 was mostly implemented from the ground up in earlier versions of ROS. The main custom software components we analysed for AgRob V16 consist in high-level controllers and path planners for slope terrain. Lastly, the coding styles of both robots are vastly different, despite both being programmed

in C++. TurtleBot2 was made with open source in mind, and it has received contributions and reviews from the community over the years, while AgRob V16 has not. In order to fully resolve the extracted model for AgRob V16 (see Fig. 8b), we had to provide a total of 13 hints (21 lines of YAML), where 4 are tied to a Python GUI node, which HAROS cannot parse.

For both case studies, we have checked – through manual code inspection and runtime inspection with ROS tools – whether the extracted models match and include all Resources and Links they are supposed to. We have observed that, in general, the developed tools are capable of finding most occurrences of ROS Primitive Links and the visualiser is capable of identifying candidates for unresolved topics. The major limitations, for the current version, are extended versions of the primitives provided by packages such as *message_filters* and *tf2*. The former provides utilities, for instance, to synchronise subscriptions, i.e. triggering a callback function when multiple messages are available. The latter builds an entire new graph of relations between coordinate frames, i.e. it can keep track of the robot's pose in the world, but also of a gripper's pose relative to the robot's base. Multiple nodes read and alter these frames, which makes the *tf2* tree an indirect communication channel. Even though the basic publisher-subscriber primitives are at the bottom of these extensions, they provide a different library interface that requires adjustments to the extractors. This is the main reason why, even in the resolved version, AgRob V16's model has some disconnected subgraphs – they interact mostly with the *tf2* structure.

B. Example Queries

In order to test our query engine, we have implemented the following catalogue of queries, based on common practices in ROS development.

1) Are global ROS names used?
2) Are there any conditional publishers or subscribers?
3) Do message types match on both ends?
4) Are there any unbounded message queues?
5) Are there any message queues of size 1?
6) Are there any disconnected topics of the same type, with similar names?
7) Are all nodes (transitively) publishing to the robot base?

Global ROS names (names beginning with a slash) are unaffected by most name resolution rules of ROS, which requires additional attention to name remappings to create multiple instances of the same Node within a single Configuration. For instance, if two TurtleBot2 robots existed within the same network and they used global names to publish sensor readings, it is likely that each robot would receive sensor data from both. This leads to additional maintenance effort, which justifies the query. The use of global names was actually an issue with the source code of AgRob V16, which our tool helped solve.

Conditional publishers and subscribers can be easily spotted in the diagrams, and there is no special justification for the query, besides these constructs leading to an additional effort in understanding the architecture. In many cases, these are loops subscribing topics from a list parameter, and there is no significantly better alternative.

There is a type checking system for messages in ROS, but it is only active during runtime. When a message type mismatch occurs, a warning is given and messages of the wrong type are discarded. This lack of communication often has noticeable effects that make it evident, but bringing this feedback to compile time is simple with our query system, and is an additional step to reduce development time. Fig. 9 illustrates how to implement type checking for Topics (i.e., publisher-subscriber only). The query code can be roughly read as: *for all Configurations, find all pairs of publishers and subscribers where the Topic names match, but the message types do not*. Implementing the same type check for Services would be equally simple. We have found no matches for this query, which is to be expected from working systems.

Message queue sizes should be carefully chosen in ROS. Avoiding unbounded message queues is a given, as they could use up all the available memory. Queues of size 1 are a very particular case. They are relatively common [2], but they can lead to message loss. Whether it is an intended effect should be analysed on a case by case basis. They are not very common in TurtleBot2, but there are quite a few occurrences in AgRob V16, mostly in third-party packages.

Query 6 is slightly more complex in terms of implementation, as seen in Fig. 10, but it shows that the query language allows for some programmatic freedom. For this query, we used some heuristics based on string comparison of the Topic names of publishers and subscribers, to try and detect those where the message types match, but the names are only slightly different (e.g. the proper names match, but the namespace prefix does not). Many times, this could be an indicator that the developer applied a name remapping on one of the nodes, but forgot to match it in another node, or that they forgot to apply remappings altogether. Wrong name remappings are another issue in ROS that can be manually detected during runtime, but for which there are no built-in compile-time checks of any kind. There are matches for this query in TurtleBot2, but they are intentional.

The final query, asking whether all nodes publish transitively to the robot base, had to be adapted for TurtleBot2 and AgRob V16, since the names of the base nodes are different. For convenience, the query language features a reflexive transitive closure operation over nodes. This, more than the previous queries, shows how a one-size-fits-all query catalogue is hard to achieve, and possibly not very useful. A query catalogue is more useful when customised to the system under analysis, taking into account the system's goals and the employed coding standards. This query found a few expected matches in AgRob V16, such as the diagnostics GUI, which does not publish messages at all.

V. Related Work

Metamodels for robotic systems, and ROS in particular, have been proposed before, with various goals, modelling languages and levels of detail. Many proposals tend towards graphical modelling languages and editors [3], [4], although

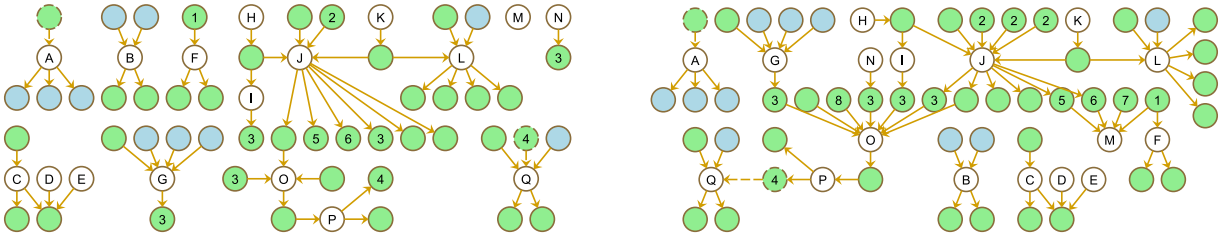

Fig. 8. Rendering of an AgRob V16 Configuration without extraction hints (a) and with user-provided hints (b).

```
for c in <configs>,
let mismatches = {
  for p in <c/nodes/publishers | c/nodes/subscribers>,
      s in <c/nodes/publishers | c/nodes/subscribers>
  where p.topic_name == s.topic_name
        and p.type != s.type
  return p, s }
return mismatches
```

Fig. 9. Query for publisher/subscriber message type mismatches.

```
for c in <configs>
let pairs = { for
  p in <c/nodes/publishers[self.topic.is_disconnected]>,
  s in <c/nodes/subscribers[self.topic.is_disconnected]>
  where p.type == s.type and
        (p.topic.id.endswith(s.topic.name)
         or s.topic.id.endswith(p.topic.name)
         or p.rosname.full == s.rosname.full
         or p.rosname.full.endswith(s.rosname.own)
         or s.rosname.full.endswith(p.rosname.own)
         or p.rosname.given == s.rosname.given)
  return p, s }
where len(pairs) > 0
return pairs
```

Fig. 10. Query for disconnected topics with matching message types but slightly different names (possibly a name remapping mistake).

some focus on standard architecture modelling languages, such as AADL [5], or on the analysis of safety requirements [6]. These are model-driven approaches, intended for use in model-first environments, where parts of source code are automatically generated. Our proposal lacks the formal specification of properties, supported by model checking, that can be found in approaches using GenoM [7] or RoboChart [8], [9], for instance. On the other hand, our metamodel is much closer to the implementation level, in terms of abstraction, and entirely compatible with a code-first development process (currently, the norm in ROS development) through static analysis.

Software Model Checking [10] is a technique that focuses on the extraction and analysis of models of the source code. Our proposal is similar in concept, but instead of building a general abstraction of the program under analysis (e.g. using finite automata) we build a domain-specific model of the ROS architecture that the source code represents (e.g. packages, nodes, topics). Having an abstraction of the program is not our goal, but a step to build the final model. Structures like *Code Property Graphs* [11] backed by query support are especially useful in this context. Unfortunately, we found their particular implementation to be too code-oriented. It is better suited for security automation, or coding standard compliance, while we aim for more expressive power over familiar ROS concepts.

Performing static analysis over distributed, dynamic systems with asynchronous message passing (as is the case with ROS) is particularly challenging. The publisher-subscriber architecture provides immense flexibility, at the cost of making it more difficult to prove correctness and predict the behaviour of a system. Despite its challenges, static analysis was shown to be useful in such systems [12], [13], by itself and as a complement to dynamic analyses. We support this view, given that some properties (timed ones, in particular) are more naturally verified at runtime. The approach presented in [13] is very similar to our own, in the sense that it aims to strike a balance between soundness and completeness, and uses a communication model of the underlying system built from control flow and data flow information. Another interesting detail, which we are also starting to explore, is the embedding of domain-specific knowledge in the tool to improve its performance. The main difference is their focus on Erlang systems and embedding of built-in functions, while we focus on ROS systems and embed entire ROS nodes and communication interfaces.

The application of static analysis tools to ROS source code is a topic of increasing research interest, but with limited examples that we are aware of. A literature review on safety certification practices [14] shows that the verification of robots implemented in formal specification languages is quite more common. In [15] the authors present an implementation for a collision avoidance safety function in C, annotated with formal contracts and following the MISRA C coding standard. The algorithm's correctness is verified using a theorem prover. We see this as an ideal development process for the expert software engineer, not for the average roboticist. Besides, we aim to provide support mostly for architecture-related properties, with minimal (if any) specification requirements.

PhrikyUnits is a static analysis tool to detect inconsistencies with physical units in ROS source code (e.g. adding meters to meters squared), which has been used to study a large sample of repositories [16]. While this is a common issue with exchanged ROS messages, for instance, we focus on properties more closely tied to the integration aspects of the system, such as guaranteeing that nodes are correctly configured.

Witte and Tichy [17] share our objective of analysing ROS software architectures and configurations, to issue warnings and detect possible errors. Their approach also includes a tool to

extract information from launch files, although conditionals and name remappings are not implemented. The main difference to our approach is that they propose dynamic analysis, running one node at a time within a sandbox, to intercept calls to the ROS primitives. This may be more accurate, but it makes it hard to deal with multiple execution paths. Static analysis, on the other hand, is able to detect variability points.

Our previous work proposed HAROS, a framework for static analysis of ROS software [18]. In its first iteration, HAROS focused on basic quality metrics and compliance with coding style guides. It was already able to extract ROS packages and explore their internal structure, despite no proper metamodel being available at the time. Using feature diagrams, we presented a characterisation of the main features and primitives of ROS [2]. Our current metamodel is a refinement of this characterisation. The same work had already implemented prototype analysis tools that extracted usage statistics for the various ROS primitives, leveraging the plug-in infrastructure of HAROS. Such plug-ins laid out the basis for our current C++ and launch file parsers.

VI. Conclusion

This work aims to promote the comprehension of ROS systems at static time by proposing a metamodel to describe the architecture of ROS software systems, along with tools that scan ROS source code to build, visualise and query the corresponding models. These help users validate and reason about the design of ROS systems at static time, in a way that would be infeasible to replicate through source code inspection. The automatic and timely feedback, graphical and otherwise, is valuable during development, and helps ensure that certain design safety rules hold. Since complete model extraction from source is impossible in general, the metamodel supports variability points, which are natively processed by the tools. Alternatively, users may provide extraction hints, which also instils specification habits – a core step towards the application of more sophisticated analyses to check the correctness of a ROS application, such as Software Model Checking techniques to ensure code safety.

We are currently applying the framework to industrial-level ROS systems and extending it to support additional ROS features (e.g. Actions) and context-specific analyses (e.g. the *tf2* package). In the near future we intend to extend our technique to handle ROS systems with many different configurations, by incorporating analysis techniques previously developed for Software Product Lines, enabling the user to visualise and analyse the multiple variants of its robotic system at once. Lastly, we are also looking into replicating our extraction process for Python ROS code, as in high-level planners or non-critical code it tends to dominate over C++.

Acknowledgment

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583).

References

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. [Online]. Available: https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf

[2] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos, "Mining the usage patterns of ROS primitives," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3855–3860.

[3] P. S. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar, "ROSMOD: A toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS," in *2015 International Symposium on Rapid System Prototyping (RSP)*, Oct 2015, pp. 39–45.

[4] S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel, "RAFCON: A graphical tool for engineering complex, robotic tasks," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 3283–3290.

[5] G. Bardaro and M. Matteucci, "Using AADL to model and develop ROS-based robotic application," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, April 2017, pp. 204–207.

[6] V. Gribov and H. Voos, "Safety oriented software engineering process for autonomous robots," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2013, pp. 1–8.

[7] M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet, "Model checking real-time properties on the functional layer of autonomous robots," in *Formal Methods and Software Engineering*. Springer International Publishing, 2016, pp. 383–399.

[8] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis, "Automatic property checking of robotic applications," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3869–3876.

[9] P. Ribeiro, A. Miyazawa, W. Li, A. Cavalcanti, and J. Timmis, "Modelling and verification of timed robotic controllers," in *Integrated Formal Methods*. Springer, 2017, pp. 18–33.

[10] G. J. Holzmann and M. H. Smith, "Software model checking: Extracting verification models from source code," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 65–79, 2001.

[11] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 590–604.

[12] D. Ganesan, M. Lindvall, L. Ruley, R. Wiegand, V. Ly, and T. Tsui, "Architectural analysis of systems based on the publisher-subscriber style," in *17th Working Conference on Reverse Engineering, WCRE 2010*, 2010.

[13] M. Christakis and K. Sagonas, "Detection of asynchronous message passing errors using static analysis," in *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2011, pp. 5–18.

[14] 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.

[15] H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter, "Guaranteeing functional safety: design for provability and computer-aided verification," *Autonomous Robots*, vol. 32, no. 3, pp. 303–331, April 2012.

[16] J. P. Ore, S. Elbaum, and C. Detweiler, "Dimensional inconsistencies in code and ROS messages: A study of 5.9M lines of code," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2017, pp. 712–718.

[17] T. Witte and M. Tichy, "Checking consistency of robot software architectures in ROS," in *2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE)*, 2018, pp. 1–8.

[18] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 4491–4496.