Checking consistency of robot software architectures in ROS

Thomas Witte and Matthias Tichy
Ulm University
Software Engineering and Programming Languages
thomas.witte,matthias.tichy@uni-ulm.de

ABSTRACT

Context: The software architecture of complex robot systems is usually divided into components. The software is then the configuration and combination of those components and their connectors.
Objective: In the Robot Operating System (ROS), this architectural configuration, the ROS node graph, is partly defined in code and created at run-time. The static information about the architecture in the configuration is limited and checking the consistency at development time is not possible. The full software has to be manually executed to check the consistency and debug configuration errors.
Method: We propose an approach and a corresponding tool to analyze ROS nodes and their launch files to check consistency and issue warnings if potential problems are detected. The approach uses both static analysis of the launch files as well as dynamic analysis of individual ROS nodes to reconstruct the node graph without executing the whole launch configuration. The nodes are executed in a sandbox to prevent side effects and enable the integration of the analysis tool, e.g., into automated testing systems.
Results: The evaluation on internal and publicly available ROS projects shows that we are able to reconstruct the complete architecture of the system if the nodes implement a common lifecycle.
Conclusion: The approach enables ROS developers to avoid creating incompatible architectures and check consistency already at development time. The approach can be extended to also monitor architectural consistency at run time.

CCS CONCEPTS

• **Computer systems organization** → *Maintainability and maintenance*; • **Software and its engineering** → *Formal methods*; *Architecture description languages*;

KEYWORDS

ROS, roslaunch, software architecture description, dynamic analysis

ACM Reference Format:
Thomas Witte and Matthias Tichy. 2018. Checking consistency of robot software architectures in ROS. In *RoSE'18: RoSE'18:IEEE/ACM 1st International Workshop on Robotics Software Engineering , May 28 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3196558.3196559

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RoSE'18, May 28 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5760-9/18/05…$15.00
https://doi.org/10.1145/3196558.3196559

1 INTRODUCTION

The Robot Operating System (ROS, [13]) is a widely used framework for developing robotics applications. Its Open Source license and the active community lead to broad adoption throughout many robotics domains [5]. ROS applications consist of multiple nodes that communicate in a peer-to-peer fashion with a rosmaster functioning as a name server and parameter database. The network is designed to be flexible and reconfigurable at run-time. Nodes can be started at arbitrary times and join or leave the network without interrupting or affecting its function.

Enabling such highly dynamic modifications on a running application greatly accelerates development speed and improves the fault tolerance of the system but may also introduce quality issues. The resulting application's components possible combinations and states are impossible to test exhaustively. Due to the dynamic nature, almost no static guarantees, like compatibility of interfaces, can be made at compile time. This hinders the adoption of ROS in commercial or production systems.

To facilitate starting and configuring multiple nodes at once, ROS ships with the tool roslaunch [4]. Roslaunch defines an XML schema for its configuration files that allows to describe all or parts of the ROS application. This configuration mainly consists of required nodes, their namespace hierarchy, configuration, launch arguments, and distribution over available host computers. When executed, roslaunch simply starts each node with its respective configuration in no specific order.

This configuration format can be seen as a rudimentary architecture description language [8] for ROS applications as it models the composition and configuration of components in the application as well as structuring elements like namespaces and hosts. It is, however, incomplete: communication channels are not explicitly modeled. The two main communication forms in ROS – topics for publish-subscribe style broadcasting of messages and services for remote procedure calls in other nodes – are not described in a declarative way but created programmatically by the nodes at run-time.

1.1 Problem Statement

ROS nodes connect implicitly through named topics or services. This can lead to an application that fails at run-time due to mistakenly unconnected components or unintentional and unwanted connections. If a node is misconfigured, it may refuse to start. In all of these cases, the application might fail silently. Per default, the ROS network keeps running and does not try to restart crashed nodes[1]. No warning is issued if a topic has subscribers but no publisher.

[1]Roslaunch can be configured to automatically restart crashed nodes. However, simply restarting the node oftentimes just hides the underlying problem and causes bugs that are even harder to find and resolve.

Errors in roslaunch configurations are therefore hard to detect – even at run-time [1]. Especially, roslaunch configurations for live operation of robots are hard to test safely. The robot has to be started while debugging the configuration. Reusing parts of the configuration between simulation and live operation, creating special launch files to test the interface of a node, debugging running configurations as well as following best practices and using idioms for roslaunch configurations are common practice to mitigate these problems [6].

During development, launch configuration "age" particularly fast as the topic / service interface and configuration keys often change. If these configurations are not executed regularly, they soon become unusable. Testing configurations is done manually most of the time in particular for live robot operation. From our experience, best practices and idioms can hardly be enforced and changes to the interface of a node during development are often not applied to all roslaunch configurations. Sometimes, testing complex applications spanning multiple hosts is impossible on development machines.

1.2 Research Questions

As a first step towards solving the aforementioned problems, we aim to check launch configurations for common inconsistencies in a safer way – without actually starting and endangering the robot and its operator. The following three research questions guided our development of a new approach towards checking ROS architectures.

- RQ 1: What information can be derived statically from launch files?
- RQ 2: How can we retrieve the missing architectural information without executing the launch file?
- RQ 3: Which of these sources of information are reliable enough to reconstruct the ROS network graph?

The contribution of this paper is an approach to analyze roslaunch configurations safely and reliably by first resolving all includes and substitutions in the input launch file and creating a *node tree* from the static information in the launch file. As information on ROS topics and services cannot be derived solely from the roslaunch configuration, we use multiple strategies to analyze the referenced nodes.

Analysis of the nodes' source code does not yield usable information in most cases; tests using a simple pattern matching approach on the source files could not produce any topic information as all nodes in our own projects make topic names configurable. Instead, the nodes are individually executed in a sandboxed environment. By intercepting calls to the language-specific ROS client library, we are able to extract the missing topic and service information, as long as the nodes' lifecycle follows a common structure and best practice as well as that the topics and services do not depend on run-time or sensor data. In these cases, the developer can provide the missing information through annotations in the analyzed launch files.

We developed a tool that implements most of the described approach as a proof-of-concept prototype [17]. First tests using launch files not only from our own projects show promising results. For simple examples, the run-time graph can be derived completely. Due to unimplemented features, complex launch configurations cannot be analyzed accurately yet.

After discussing existing work and tools in Section 2, we describe our approach and its limitations in Section 3 in more detail. Section 4 presents preliminary test results from our prototype tool. Open issues in our approach and the prototype tool are discussed in Section 5.

2 RELATED WORK

Architecture description languages (ADLs) are a broad and inconsistently defined research field. Medvidovic and Taylor proposed a framework [8] to classify and compare different ADLs. According to their classification, roslaunch configurations do not meet the requirements for an ADL, as they do not contain information on connectors. We aim to retrieve this missing information through dynamic analysis of the components.

Brugali and Gherardi adapted their HyperFlex framework [2] to ROS. Their framework enables defining the architecture of a robotics application constructively. Software product lines (SPLs) are used to describe possible variations in the robotics application. The consistency of the resulting ROS graph is guaranteed through constraints on the configuration of the SPL. In contrast, our approach tries to deduct the architecture from existing launch configurations to check the consistency of the resulting node graph.

To configure and start multiple ROS nodes together, roslaunch is the standard tool available in every ROS installation. Other implementations exist that use the same configuration format but offer additional features: rqt_launch [14] has a graphical user interface to start, stop or restart individual nodes and can start all configured nodes in a deterministic order. node_manager_fkie [16] specializes in starting nodes across multiple hosts. It can connect to multiple rosmaster instances and monitor running nodes.

Multiple graphical editors for ROS launch files were proposed. However, these projects either aim to visualize launch file structure and not the resulting run-time graph or need additional files that describe the resulting run-time behavior.

rxDeveloper [10] tries to simplify the creation of launch files by visualizing the ROS graph structure. It uses *specification files* to get information on topics, services and parameters of the configured nodes. Another graphical launch file editor was developed as part of the RADOE project [11] since development of rxDeveloper seems to have halted. Here, the user must configure topic publications and subscriptions by hand before connecting multiple nodes.

Viki [7] is a graphical ROS interface that improves upon previous attempts to provide a data flow based GUI by introducing *modules*. These *modules* provide an abstraction over the fine granularity of ROS nodes and group multiple ROS nodes into function blocks, that can be connected through input and output ports. The resulting architecture graph can be exported in the roslaunch configuration format or started directly. Viki depends on module descriptions that provide the necessary information on input and output ports. In contrast, our approach tries to derive similar information directly from the respective ROS node.

Similar to our tool prototype, rqt_launchtree [15] statically analyzes roslaunch configurations. It resolves includes and substitution parameters and offers a GUI to trace nodes back to the corresponding configuration files. Some inconsistencies trigger warnings in the GUI, like parameters set twice to different values. However, all

consistency checks are limited to the information provided in the launch files. Misconfigured topics and services can not be detected.

ROS provides the tool roswtf [3] to search for problems in a running ROS application. It uses a set of rules to detect multiple common problems like subscribed but unpublished topics. Consistency checks on the ROS graph are done during run-time. Additionally, nodes that do not exist but are referenced in launch configurations, duplicate nodes and missing includes can be found statically.

Bihlmaier et al. proposed ARNI [1], a framework to monitor large ROS systems at run-time to find performance bottlenecks and configuration errors. If violations of a known good state are detected, automatic countermeasures can be taken to ensure continued functionality of the ROS network.

Rostest is an extension to roslaunch. The roslaunch configuration serves as a fixture for testing ROS nodes. A special *test* tag can be used to start a test node that integrates common test frameworks, such as *gtest* for C++ or *unittest* for Python. Unlike our prototype tool, rostest can not test roslaunch configurations themselves but merely uses launch configurations to describe the test environment for nodes.

The problem of inconsistent and misconfigured ROS applications has been identified and described before. The proposed solutions focus on either guaranteeing the consistency constructively by constraining the composition of nodes or checking and monitoring the ROS graph at run-time for problems. In contrast, our approach does not constrain launch files and can be used on existing configurations without the need to actually execute the configuration and running into these problems.

3 OUR APPROACH

Roslaunch configurations describe the architecture and parametrization of the system. However, they lack information on message channels, such as topics and services between the different nodes. It is therefore possible to retrieve the *node tree*, the nodes and their enclosing namespaces, which forms a tree structure but not the topic and service connections between these nodes.

Due to the dynamic nature and lack of enforced lifecycle of ROS nodes, topics and services can be created (and removed) at any time and depending on parameters, arguments the environment and even sensor data or data received through ROS. It is therefore impossible to get complete information on the topics and services a node provides or uses solely through static analysis.

Our approach uses a combination of multiple sources to retrieve additional information to decorate the node tree: currently, only *sandboxed execution* and *launch file annotations* are implemented but other sources, such as static analysis of the node's source code can be easily integrated and improve the quality and completeness of the annotated node tree.

Multiple report plugins can then use the annotated node tree to perform e.g. consistency checks or visualize the node graph. These reports can in turn be used as input for other tools. A CI system can use the consistency reports to detect and reject changes that break existing launch configurations. The expected run-time graph, our tool produces, can be compared with the specified architecture or a known good graph from a previous run.

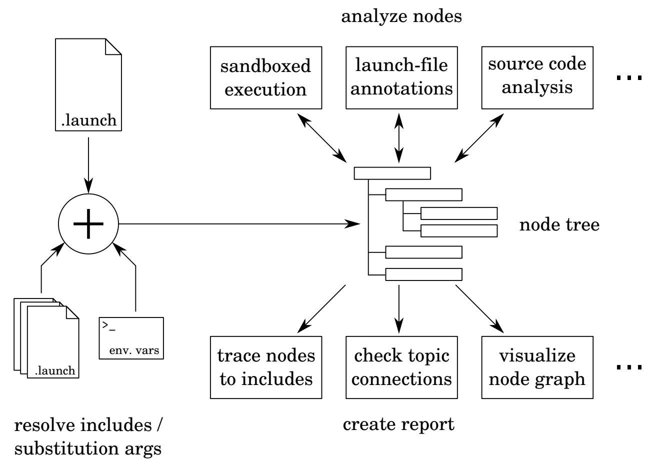

Figure 1: Schematic overview of the analysis toolchain. Node information that cannot be retrieved from the launch files directly, is provided by one or more analysis plugins.

3.1 Node tree creation

As a first step in the analysis of a roslaunch configuration, the XML-Structure must be validated against its XML Schema.

Roslaunch allows parametrization of any string in the XML-Structure through *substitution args*, which must be resolved prior to any subsequent analysis while parsing the configuration. Additional launch files – whose file names might need to be resolved – can be included into the launch configuration, which might require processing additional substitutions and inclusions recursively. These substitutions might depend on environment variables, installed ROS packages, the start arguments for the launch configuration, the directory structure or any Python expression. Therefore, our prototype implementation depends on an execution environment as close as possible to the launch environment and spawns the system's python interpreter to evaluate *eval* substitutions.

A tree visitor can then collect all node tags in the XML structure and create the node tree. Inner nodes are ROS namespaces and leaves are ROS nodes. Additional information from the launch files is saved in the leaf nodes: start arguments, parameters, name and remappings might influence the run-time behavior of the ROS node and are necessary for the node analysis. Tracing information, like the launch file name corresponding to the ROS node can improve the reporting quality and is saved as well (Figure 2).

However, further information on Topics and Services cannot be retrieved from the launch configuration. At run-time, nodes can freely communicate over any topic or service by its name. To retrieve this missing information, the nodes referenced in the roslaunch configuration must be analyzed. For each node, the started binary, its parameters and configuration are known through the launch configuration.

Neither static nor dynamic analysis techniques can obtain complete topic and service information for a node. Listing 1 shows a minimal example of a node that publishes to a random topic. A launch configuration using this node might be inconsistent during

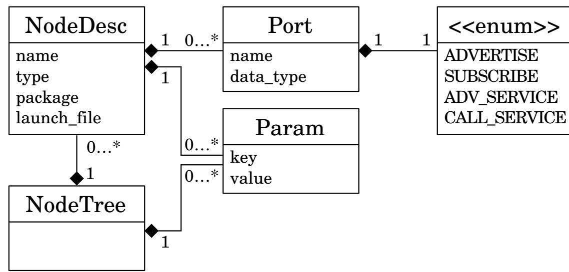

Figure 2: Data model of the node tree.

some launches but consistent during others. While a ROS node publishing to a random topic might not exist in real world applications, other more common nodes can show a similar behavior: *rosbag play* nodes create topics depending on the content of the recording and camera tracking nodes often create topics depending on the currently visible or tracked objects.

Listing 1: The ROS node publishes to a random topic.

```
#!/usr/bin/env python

import rospy
import random
from std_msgs.msg import String

if __name__ == '__main__':
    topic = 'topic'+str(random.randint(0,5))
    rospy.init_node('rnd_node')
    pub = rospy.Publisher(topic,
                          String,
                          queue_size=10)
    pub.publish('hello')
    rospy.spin()
```

3.2 Node source analysis

The natural extension to the static launch file analysis, is to statically analyze the source code of the ROS node.

While this probably works for scripting languages like Python, this introduces multiple severe limitations for nodes written in compiled languages like C++. The launch file references the node executable; finding the corresponding source code is non-trivial. 3rd party nodes might be distributed in compiled form without locally available source code. Locating local ROS workspaces and analyzing the cmake build files to identify the source files is hardly feasible.

Alternatively, most compilers leave source file paths in the binary file – even in release builds. These can be found by searching for strings matching file name patterns in the binary. In our tests using gcc 5 the .cpp file for each translation unit can be found but none of the included header files. Finding the header files would again require analyzing the complete build files.

If the referenced files exist in the file system, they can be searched for statements that create or connect to topics or services. The scope of the source code analysis is further limited, as most C preprocessor defines and variables can not be resolved due to missing include files. The name of the topic or service must be a literal in the statement. False positives, e.g., topic subscriptions that are hidden behind conditional statements, can not be prevented, as symbolic execution techniques are limited due to the missing header files.

3.3 Sandboxed execution

Dynamic analysis techniques are more broadly applicable: compiled ROS nodes can be analyzed without access to the source code. By executing the ROS node with the parameters given in the launch configuration, only the relevant code paths are analyzed and the result is more accurate.

The *sandboxed execution analysis* exploits our observation that – despite the flexibility of ROS node – most nodes follow the simple lifecycle shown in Figure 3. In this case, the topic and service connections are created after the configuration and arguments are processed, but before entering the event loop and do not change during the run-time. Particularly, topic and service connections are independent from run-time data received through ROS or sensor data.

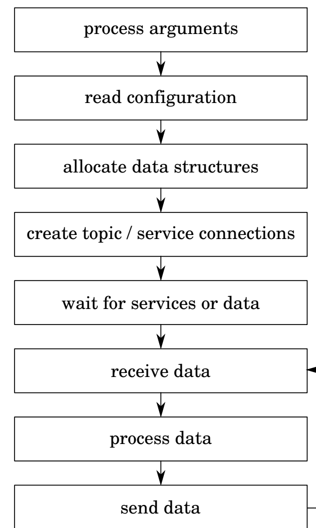

Figure 3: Common lifecycle of a ROS node.

Furthermore, it is assumed that most ROS nodes act independently and do not need a combination of other nodes to function. Nodes communicate with each other using only ROS communication channels and create their input and output topics before checking for the existence of services and topic publishers. A notable exception to this is *gazebo*. The *gzserver* does not expose topics itself but is needed by *spawn_model*.

A ROS node following this lifecycle can be analyzed by starting it with the given configuration and arguments and then observing which topics and services it subscribes or publishes. As the connections do not change after entering the main loop, the started node can be killed again after a few seconds. Figure 4 shows a schematic representation of the dynamic analysis approach used.

The execution of the node is sandboxed using Firejail[12] to prevent any persistent changes to the file system. Stronger isolation mechanisms such as docker might limit the executed node even further but complicate the replication of the execution environment including the ROS stack and available system libraries. An application sandbox on the other hand is designed to preserve the host environment while limiting some capabilities like file system access.

In order to gather information on published or subscribed topics and services, an additional library is injected into the node process to intercept the corresponding calls to the roscpp or rospy libraries. This approach offers some benefits over querying the rosmaster. The ROS master API has no function to query the request and result data types of services and has no information on service calls: only announced services are known to the rosmaster, clients call these services directly in a peer-to-peer fashion. By intercepting the library calls, this additional information is still available. The response and results can even be manipulated to mock non-existing services or avoid blocking if the node waits for a service or topic to become available.

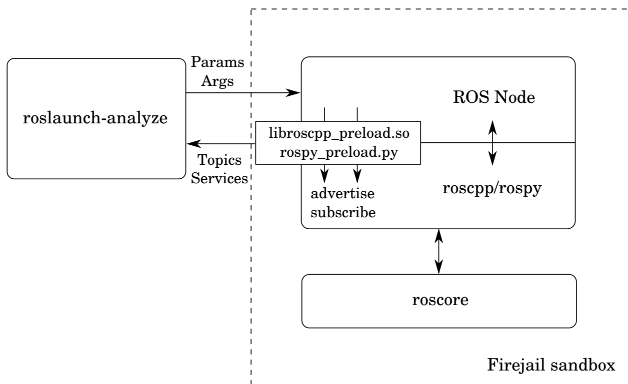

Figure 4: An additional library intercepts calls to the ROS library and provides topic and service information. Sandboxed execution ensures that no persistent writes to disk are possible.

Most ROS language bindings use a specific client library. The two most used client libraries are roscpp for nodes written in C++ and rospy for python nodes. Both libraries themselves are implemented in their respective language, and therefore need a specific library and technique to intercept library calls.

To intercept calls to the *rospy* python library, aspect oriented programming using *aspectlib* is used. Wrappers around the intercepted functions are weaved into rospy before starting the actual node. Listing 2 intercepts calls to the constructor of `rospy.Publisher`.

Unfortunately, this powerful and safe approach is not possible in C++ without recompilation of the code. For *roscpp*, an additional library is preloaded using the dynamic linker. The library contains symbols for the intercepted methods that shadow the symbols in the original roscpp library. All calls are logged and the original method is called to ensure unaltered functionality. Listing 3 shows the wrapper for the advertise method. It uses *dlsym* to load the

Listing 2: Function to intercept calls to the Publisher constructor in the rospy library.

```
@aspectlib.Aspect
def wrap_publisher(self, topic, datatype,
                   subscriber_listener=None,
                   tcp_nodelay=False,
                   latch=False,
                   headers=None,
                   queue_size=None):
    log('<<advertise>>␣'
      + rospy.names.resolve_name(topic) + '␣'
      + datatype.__module__.split('.')[0] + '/'
      + datatype.__name__)
    yield

aspectlib.weave(rospy.Publisher.__init__,
                wrap_publisher)
```

original method from the roscpp library using the mangled method name. The C++ Standard forbids casts between *void** and pointers to member functions. A union *ptm_cast* is used to defeat the type system. This rather fragile and compiler dependent code is necessary, as *dlsym* does not support C++ directly. Templated or inlined functions and methods can not be intercepted using dlsym. These functions are generated on demand while the node is compiled and no library call is necessary. Roscpp typically just creates a properties object in the templated methods and forwards the call to a non-template method that can be intercepted. Some type information on service calls is lost, as only the service type's md5 sum is forwarded in the library call. It is still possible to check whether the type of the advertised service matches the service call by comparing the md5 sums. Getting a human-readable name of the type of a called service is possible by building a lookup table from all advertised services, as a valid service call must refer to an advertised service in the configuration.

Currently, the sandboxed execution analysis only supports roscpp and rospy. Other client libraries need custom methods to intercept calls and log topic and service connections. However, ROS2 uses a common core library for all language bindings which mitigates this drawback.

3.4 Launch file annotations

If all of the aforementioned analysis plugins fail to provide the missing topic and service information, the interface of the ROS node can be specified directly in the launch file. Comments inside of node tags are parsed and may contain a topics or services tag. Listing 4 shows an example of a topics tag.

As these XML tags are inside a comment, roslaunch currently just ignores them. The surrounding comment tag is not necessary, as roslaunch just issues a warning if it encounters an unknown tag and skips it. This way, future versions of roslaunch or a watchdog node might adopt this schema extension and ensure the specified topics and services are indeed available at run-time.

Listing 3: Function to intercept calls to the advertise method in the roscpp library.

```
Publisher NodeHandle::advertise(
                AdvertiseOptions& opts)
{
    typedef Publisher (NodeHandle::*advertise_t)
                          (AdvertiseOptions&);

    static advertise_t orig_advertise = nullptr;
    if (!orig_advertise) {
        // load the symbol from the roscpp library
        // and cast it to a member pointer
        ptm_cast<advertise_t> tmp;
        tmp.pmember = nullptr;
        tmp.vs.pvoid = dlsym(RTLD_NEXT,
            "_ZN3ros10NodeHandle9advertise"
            "ERNS_16AdvertiseOptionsE");
        orig_advertise = tmp.pmember;
    }

    // get the node's namespace
    std::string ns = getUnresolvedNamespace();
    if (!ns.empty())
        ns.append("/");

    // log the topic name and data type
    log << "<<advertise>>␣" << (ns + opts.topic)
        << "␣" << opts.datatype << std::endl;

    // forward the call to the original method
    return (this->*orig_advertise)(opts);
}
```

Listing 4: Example of a topics tag that provides the missing topic data for a node.

```
<node name="listener"
      pkg="roscpp_tutorials"
      type="listener">
  <!--
  <topics>
    <topic name="chatter"
           type="String"
           class="sub"/>
  </topics>
  -->
</node>
```

3.5 Reporting

Our current prototype outputs two reports as a result of the analysis. A representation of the node tree which shows the complete namespace and node structure after all includes and substitutions. Each node and namespace is colored corresponding to the configuration file it stems from in order to allow easy tracing.

Additionally, all known topic connections are reported. For each topic, a list of known publishers and subscribers is shown to easily spot unpublished but subscribed topics.

4 EXPERIMENTAL RESULTS

A prototypical implementation was used to evaluate our approach. Launch configurations from three different projects were analyzed and compared to the run-time information *rostopic* yields when the complete configuration is started.

To test the basic functionality, we analyzed the launch configurations from the ros_tutorials repository. These launch files are example files that start small tutorial nodes. While not overly complex, they showcase different implementation techniques and a broad range of functionality for nodes using roscpp and rospy. Many nodes follow these nodes' general structure as they serve as examples in the ROS tutorials.

As shown in Table 1, for the rospy tutorials, we analyzed the provided launch files and found that for 6 out of 8 launch files the analysis yielded correct and complete topic and service information for all nodes. The remaining cases failed because of unimplemented features in our prototype, namely incorrect handling of topic renames and parameters which are not forwarded to the sandboxed execution.

The roscpp tutorials in most cases do not provide accompanying launch files for the example nodes. We therefore started each node using the sandboxed execution and compared the resulting topic and service information with the expected topics and services from a manual source code inspection. The analysis yielded correct and complete information in all 20 cases.

Table 1: Correctness of the analysis results on different launch configurations from the rospy_tutorials project.

rospy_tutorials	notes
talker_listener	✓
headers	✓
listener_with_user_data	incorrect topic renames
listener_subscribe_notify	✓
parameters	missing parameter
connection_header	✓
on_shutdown	✓
advanced_publish	✓

To test whether our approach is applicable to real world applications, we tested our prototype implementation on launch configurations from our own projects and the hector quadrotor project [9]. Correct and complete topic and service information could be derived for most nodes.

Table 2 shows the correctness of the analysis of one of our projects' launch files. For each node that is configured in the launch file, the number, name and type of topics and services found by the prototype is compared to the run-time information gathered by rostopic when the full launch configuration is executed[2]. The

[2] At startup, each node connects to the /rosout topic and advertises two services to send logging data and control its verbosity. Nodes therefore advertise at least one topic and two services which are included in the table

node lifecycle in Figure 3 is strongly encouraged for our own code and our guidelines for launch configurations advise against the use of remappings or non-private parameters if possible. Therefore, no missing feature of the prototype implementation is hit and the nodes satisfy the requirements for successful analysis introduced in Section 3.3. The node and namespace structure is correctly derived from the launch file and all node connections have been retrieved using the sandboxed execution approach.

Table 2: Correctness of the analysis of a launch file from our own projects.

collision_test_rviz	topics	services
drone1/quad_node	4/4	2/2
drone1/quad_script_node	9/9	2/2
drone1/sim_photo_node	4/4	3/3
drone1/trajectory_client_node	11/11	2/2
drone2/quad_node	4/4	2/2
drone2/quad_script_node	9/9	2/2
drone2/sim_photo_node	4/4	3/3
drone2/trajectory_client_node	11/11	2/2
rviz	8/8	3/3
trajectory_server_node	7/7	2/2

If the launch configuration contains mainly 3rd-party nodes, the accuracy of our analysis decreases. Table 3 shows analysis results for the hector indoor slam demo configuration from the hector quadrotor project. We could correctly identify topics and services for 6 of the 14 running nodes.

Table 3: Correctness of the analysis of a launch file from the hector_quadrotor project.

hector/indoor_slam_gazebo	topics	services
gazebo	9/89	30/64
robot_state_publisher	2/5	2/2
ground_truth_to_tf	6/6	2/2
pose_estimation	26/-	2/-
controller_spawner	0/2	0/2
estop_relay	3/3	2/2
pose_action	4/8	2/2
landing_action	8/15	2/2
takeoff_action	8/11	2/2
spawn_robot	2/-	2/-
hector_mapping	13/13	3/3
hector_trajectory_server	4/6	2/4
hector_geotiff_node	3/3	2/2
rviz	13/13	3/3
joy	4/4	2/2
teleop	17/18	2/2

The node tree – built from the statically available information in the configuration file – was mostly correct. However, our prototype currently does not support the *if* and *unless* attributes for, e.g., nodes or groups and analyzed the pose_estimation node that is not started when the configuration is executed under roslaunch. The spawn_robot node exits as soon as it spawned the robot model in gazebo, so its topic and service connections were not recorded by rostopic. Due to incompletely forwarded configuration in our prototype, the robot_state_publisher and spawn_robot nodes did not start correctly.

Some nodes such as pose_action, landing_action and takeoff_action do not follow the aforementioned lifecycle and wait for data on topics before finishing the initialization of all topics and services. This causes the sandboxed analysis to yield incomplete results as the nodes wait for data and block until the nodes are terminated. Our prototype can not yet send dummy data to the detected topics to continue execution in these situations.

Gazebo does only exhibit a fraction of its topics and services, as the robot model is not loaded when run in isolation. The controller_spawner and spawn_robot nodes are closely connected to the gazebo simulator and do not expose their full functionality if they are started separately.

5 OPEN ISSUES

Missing features in our prototype implementation. Our prototype is still in development and is still missing some crucial features. The static analysis of the launch files is still incomplete and multiple tags are completely ignored. Handling of inline YAML and loading of parameter files is not yet implemented and global parameters are not handled correctly in all cases. Machine tags, environment tags, remappings and attributes to conditionally enable or disable tags in the launch file are currently completely ignored.

Analysis of nodes that implement different lifecycles. To analyze topics and services exposed by nodes, our prototype implements only the sandboxed execution presented in Section 3.3 and launch file annotations (Section 3.4).

Nodes that do not implement the lifecycle shown in Figure 3 are in most cases not correctly analyzed. By intercepting calls to wait for services or data and returning immediately we could work around some of these problems.

Alternatively, the analysis tool can connect to every topic and service it detects and provide it with dummy data to skip any wait statements in the analyzed node.

Waiting until necessary topics or services are announced is a common pattern to avoid race conditions at startup due to the non-deterministic order in which the nodes are started by roslaunch. Checking for nodes and services before entering the main loop can also prevent erroneous behavior in case of misconfiguration. However, it is better to do such checks after all topic and service connections are created to avoid deadlocks in the node graph (cf. Figure 3).

Nodes that cannot be analyzed using sandboxed execution. If a node's interface depends on run-time data – e.g. the topics, it publishes to correspond to the names of tracked objects – the node does not expose all topics and services when executed in a sandbox.

Similarly, a driver node that connects to hardware that is not present during the analysis might exit or crash before initializing its topics and services, which prevents successful analysis.

It is possible to detect crashes or prematurely exiting nodes and adapt the analysis strategy accordingly. Launch file annotations for these nodes could be automatically generated from executing other launch files that use the same nodes.

Dependent nodes. Nodes can configure other nodes by communicating over other channels than ROS topics and services, e.g. plugin interfaces or send special messages to other nodes that cause these nodes to create topics. Such dependent nodes can not be started in isolation but must be started as a group of nodes to be successfully analyzed.

It must be ensured, that the group of simultaneously started nodes is minimal, as starting multiple nodes violates the isolation principle of the sandboxed analysis.

6 CONCLUSION AND FUTURE WORK

We presented an approach to analyze roslaunch configurations by first statically analyzing the roslaunch configuration itself and then retrieving missing information from the ROS nodes themselves by executing them in isolation in an application sandbox. By injecting an analysis library into the node process, that intercepts calls to the language-specific core ROS library, we are able to retrieve more information than is available to the rosmaster.

RQ 1: What information can be derived statically from launch files? Roslaunch files contain the configuration of the nodes that comprise the system. This includes the namespace structure, parametrization and parameters to the nodes. By incorporating the environment and the arguments, the configuration is executed with, all substitutions in the launch file can be resolved. However, the launch file does not contain information on communication channels, as ROS connects to topics and services programmatically at run-time and not in a declarative fashion.

RQ 2: How can we retrieve the missing architectural information without executing the launch file? We developed a dynamic approach to retrieve the missing topic and service information directly from the configured ROS nodes. The connections of a ROS node are created at run-time and may depend on data at run-time but are in most cases static once configured at startup. The nodes are executed to initialize in a sandbox and the library calls that create topic or service connections are intercepted and logged.

Alternatively, if the node can not be analyzed using the sandboxed execution approach, the launch files can be annotated to provide the missing architectural information.

RQ 3: Which of these sources of information are reliable enough to reconstruct the ROS network graph? Our tool prototype is able to correctly reconstruct the ROS network graph if the nodes follow the lifecycle model in Figure 3. Other nodes sometimes yield incomplete information but the missing information can be annotated in the launch file to enable our tool to reliably reconstruct the ROS network graph.

6.1 Future Work

We will continue to improve the tool prototype to address the remaining issues from Section 5. Once the tool supports all roslaunch features, we plan to conduct an extended evaluation on a broader set of projects.

The reporting functionality can be extended and integrated into development toolchains. Automatic regression testing of launch configurations in a CI environment can detect interface changes of nodes that are not correctly propagated to the launch files. Once a launch configuration is started correctly, it is not guaranteed to operate correctly, as nodes can fail or crash at run-time. A watchdog at run-time can continuously compare the current ROS graph to the expected graph generated by our analysis tool.

With the release of ROS2, our tool needs to be adapted to the planned new roslaunch configuration format which is scheduled to be released in summer 2018. With ROS2, nodes can implement a predefined lifecycle that simplifies the dynamic analysis of the nodes to retrieve topic and service information. The XML based launch file format will be replaced by python scripts to enable the use of more complex logic.

ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) as part of the DFG Priority Programme 1593 (SPP1593) (grant numbers TI 803/2-2 and TI 803/4-1).

REFERENCES

[1] Andreas Bihlmaier, Matthias Hadlich, and Heinz Wörn. 2016. Advanced ROS Network Introspection (ARNI). In *Robot Operating System (ROS)*. Springer, 651–670.

[2] Davide Brugali and Luca Gherardi. 2016. Hyperflex: A model driven toolchain for designing and configuring software control systems for autonomous robots. In *Robot Operating System (ROS)*. Springer, 509–534.

[3] Ken Conley. 2012. roswtf – ROS Wiki. (2012). http://wiki.ros.org/roswtf

[4] Ken Conley. 2017. roslaunch – ROS Wiki. (2017). http://wiki.ros.org/roslaunch

[5] Steve Cousins, Brian Gerkey, Ken Conley, and Willow Garage. 2010. Sharing software with ros [ros topics]. *IEEE Robotics & Automation Magazine* 17, 2 (2010), 12–14.

[6] Open Source Robotics Foundation. 2016. roslaunch tips for larger projects – ROS Wiki. (2016). http://wiki.ros.org/roslaunch/Tutorials/Roslaunch%20tips%20for%20larger%20projects

[7] Robin Hoogervorst, Cees Trouwborst, Alex Kamphuis, and Matteo Fumagalli. 2017. VIKI – More Than a GUI for ROS. In *Robot Operating System (ROS)*. Springer, 633–655.

[8] Nenad Medvidovic and Richard N Taylor. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering* 26, 1 (2000), 70–93.

[9] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. 2012. Comprehensive simulation of quadrotor uavs using ros and gazebo. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 400–411.

[10] Filip Müllers, Dirk Holz, and Sven Behnke. 2012. rxDeveloper: GUI-Aided Software Developmemt in ROS. *SDIR VII–ICRA* (2012).

[11] Aditya Narayanamoorthy, Renjun Li, and Zhiyong Huang. 2015. Creating ROS launch files using a visual programming interface. In *Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM), 2015 IEEE 7th International Conference on*. IEEE, 142–146.

[12] netblue30/firejail. 2014. Linux namespaces and seccomp-bpf sandbox. (2014). https://github.com/netblue30/firejail

[13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, 5.

[14] Isaac Saito and Ryan Sinnet. 2015. rqt_launch – ROS Wiki. (2015). http://wiki.ros.org/rqt_launch

[15] Philipp Schillinger. 2016. rqt_launchtree – ROS Wiki. (2016). http://wiki.ros.org/rqt_launchtree

[16] Alexander Tiderko and Timo Roehling. 2016. node_manager_fkie – ROS Wiki. (2016). http://wiki.ros.org/node_manager_fkie

[17] Thomas Witte. 2018. ros-launch-analyze. (2018). https://github.com/ThomasWitte/ros-launch-analyze