Bootstrapping MDE Development from ROS Manual Code - Part 2: Model Generation

Nadia Hammoudeh Garcia*, Ludovic Delval*, Mathias Lüdtke*, Andre Santos†, Björn Kahl* and Mirko Bordignon*
* Fraunhofer Institute for Manufacturing Engineering and Automation IPA
Stuttgart - Germany
nadia.hammoudeh.garcia, ludovic.delval, mathias.luedtke, bjoern.kahl, mirko.bordignon@ipa.fraunhofer.de
† HASLab High-Assurance Software Laboratory
INESC TEC & Universidade do Minho
Braga, Portugal
andre.f.santos@inesctec.pt

***Abstract*—In principle, Model-Driven Engineering (MDE) addresses central aspects of robotics software development. Domain experts could leverage the expressiveness of models; implementation details over different hardware could be handled by automatic code generation. In practice, most evidence points to manual code development as the norm, despite several MDE efforts in robotics. Possible reasons for this disconnect are the wide ranges of applications and target platforms making all-encompassing MDE IDEs hard to develop and maintain, with developers reverting to writing code manually. Acknowledging this, and given the opportunity to leverage a large corpus of open-source software widely adopted by the robotics community, we pursue modeling as a complement, rather than an alternative, to manually written code. Our previous work introduced metamodels to describe components, their interactions, and their resulting composition, as inspired by, but not limited to, the de-facto standard Robot Operating System (ROS). In this paper we put such metamodels into use through two contributions [1]. First, we automate the generation of models from manually written artifacts through extraction from source code and runtime system monitoring. Second, we make available an easy-to-use web infrastructure to perform the extraction, together with a growing database of models so generated. Our aim with this tooling, publicly available both as-a-service and as source code, is to lower the MDE barrier for practitioners and leverage models to 1) improve the understanding of manually written code; 2) perform correctness checks; and 3) systematize the definition and adoption of best practices through large-scale generation of models from existing code. A comprehensive example is provided as a walk-through for robotics software practitioners.**

***Index Terms*—ROS, models, development environments.**

I. Introduction

Robots are increasingly software-intensive systems. Many innovations in the field are algorithmic in nature, and thus typically implemented in software. Application domains for robotic solutions are proliferating [2], making stock hardware platforms repurposable by means of software repurposability. The wider digitization trend affects also manufacturing through flagship initiatives such as Industry 4.0, and puts an emphasis on robots and other mechatronic equipment as actors in a software-centric scenario [3]. Robotics software engineering is hence a distinctly recognized field, with dedicated publishing venues, technical committees [4], and educational curricula [5]. In the years leading up to the establishment of the field, software engineering approaches with proven track records in other domains have been applied to robotics with various degrees of technical maturity and uptake by practitioners. Among them are model-based techniques, such as model-based Software Product Lines (SPLs applied to robotics software, e.g., [6], [7]) and Model-driven Engineering Integrated Development Environments (MDE IDEs, e.g., [8]). Meanwhile, more traditional approaches broadly falling under the category of software frameworks have also been proposed [9]–[12]. Among them, the Robot Operating System (ROS [13]) gained widespread acceptance in research and service robotics, with increasing signs of adoption in the industrial domain [14]. As both developers of robot software leveraging traditional frameworks for concrete, production-grade use cases, and researchers in Model-Driven Engineering approaches to robotics software, we identified possible ways to bring some advantages of model-based techniques into standard robotics software engineering practice. The norm here being, still, to manually develop code: as the results of a user study documented in [8] reveal, ”using an integrated IDE for development is not yet standard” and ”reuse is made on the level of libraries, but very few component-based approaches are applied”. Our experience while consulting on topics centered on robotics software engineering for several organizations worldwide anecdotally confirms this finding. Aiming at completely replacing manual code development with an all-encompassing MDE IDE historically proved difficult to impossible, given the large variety of application use cases and of target platforms to support. We thus instead aim at leveraging a large existing codebase, such as the corpus of open-source ROS hand-written components (more than 3000 [15]), while using model-based techniques when possible as a complement.

The paper is structured as follows. The next section will cover related work, while the following one will summarize previous metamodeling efforts directly applicable to the contribution later presented. Sec. IV presents the first and main contribution of this paper: two approaches to automatically generate models from existing ROS software artifacts, respectively through static analysis of source code and runtime system monitoring. Sec. V presents the second contribution, i.e., a publicly available, easy-to-use cloud infrastructure to extract models from provided source code repositories, and a first database of models which we built from open-source ROS components. Sec. VI details a use case example of how our tooling can be applied to extract models from different approaches, evaluating and comparing the results and their potentials; and to ease the replacement of subcomponents by identifying common interaction patterns. The Care-O-bot 4 [16], a service robot now engineered and commercialized by the company Mojin Robotics, serves as the example. Its source code is publicly available and hence suitable for a walk-through style tutorial, while at the same time being representative of the complexity of ROS code in commercially deployed robots. Sec. VII recapitulates our contributions, and lays out some directions for future work.

II. Related work

A. ROS - The Robot Operating System

ROS [12], [13] is the software framework whose concepts we target with our effort, given its popularity and hence the impact which we can achieve among practitioners. It combines software written in common languages with little architectural constraints, and has a federated development model, leveraging common tools to easily share such components across organizations. This resulted in fast adoption and a large software ecosystem.

To distill the basic constituent entities from the system, we can resort to the commonly cited analogy of ROS being a peer-to-peer network of processes exchanging ad manipulating data (the *computation graph*). This results in ROS systems designed as a collection of small, mostly independent programs called *nodes* that run all at the same time and can communicate with each other. The communication mechanisms are the *topic* and *service* patterns. Topics are means for one-way communication and many-to-many connections, while services are used for two-ways communication and one-to-one connections. The agents of this communication are *messages* and *services*, which consist of language-independent data structures composed of primitive data types (String, Double, Int, Boolean..). Last, collections of nodes can be started through *launch files*, which can also be used to set parameters and to *remap* (i.e., reassign at runtime) names such as those of topics.

B. MDE efforts backends generating ROS code

The BRICS (Best Practices in Robotics) [17] component model (BCM) combined the model-driven approach with the separation of concerns paradigm, and introduced the 5Cs (Computation, Communication, Coordination, Configuration, and Composition) concept. The BRICS Integrated Development Environment (BRIDE [18]), bridged BCM and ROS using model-to-text (M2T) transformations to generate ROS skeleton code to be filled by an application domain expert. This achieved BRIDE's main goal of "explicit separation of two phases of the development process, i.e., capability building and the system development" [18], and showed how to fit ROS in a model-based approach, but maintained a traditional MDE top-down approach with (partial) code generation from models, not allowing the import of existing, manually developed ROS systems. BRIDE and similar efforts generating ROS "boilerplate code" through a dedicated backend leverage MDE to ease successive manual development, but do not allow for a development style intermixing manual development and e.g., model-supported checking of component composition. Related to the aim of easing the interoperability of ROS with other systems, a visual modeling language was created to transform ROS specific message to Robot Device Interface Specification (RDIS) and viceversa, by mapping the communication mechanisms [19].

The ongoing RobMoSys project [20] leverages SmartSoft [21] and promises predictable composition of both existing and new components through MDE.

C. HAROS - Static code analyzer framework for ROS

HAROS is a plug-in-based framework whose primary focus is static analysis of ROS software [22]. Its initial iteration brought general-purpose quality metrics and coding style compliance checks into the ROS ecosystem. More recent versions backed HAROS with an internal metamodel that characterizes typical source code artifacts (e.g., packages and different kinds of source files) and runtime entities in a ROS system [23] (e.g., nodes and topics). Such a metamodel, coupled with source code parsing tools, enables HAROS to reconstruct models of a ROS system via static analysis. In particular, HAROS can reverse-engineer the ROS computation graph by parsing C++ and ROS launch files. Support for other popular languages, such as Python, is ongoing work by external contributors. Extracted models can then be graphically visualized, subject to automated analyses, or exported as raw data for other applications. In summary, HAROS aims to provide support for the analysis and validation of the architectural design of ROS applications, without requiring the execution of said applications. However static analysis has well-known limitations, such as values that cannot be resolved statically. HAROS developers report this in [23], as handled by explicitly marking extracted entities as conditional or unknown, rather than raising errors, and by allowing users to aid the system in resolving some of these entities. Metamodeling targeting ROS systems are separately examined in the next section.

III. Previous Work: ROS Metamodeling

The main motivation for this work is to facilitate the composition of large systems, a task that ROS integrators perform on a regular basis. The lack of tools to validate, create deployment artifacts and test the composition at design time (currently a tedious trial-and-error verification) represents, from our point of view, one of the biggest shortcomings of ROS.

The contributions of this paper leverage and existing family of three metamodels which we developed in previous work [1]. These metamodels split the description of robot system into: 1) the ROS metamodel (the monolithic description of ROS programs), 2) the Component Interface metamodel (the extraction of the ROS-specific concepts to a generic Component-based architecture) and 3) the System metamodel (the composition of components and its connections). This previous work includes also three matching Xtext DSLs, which allow the user to check and validate against properties and constraints. All of this work is supported by an Eclipse tooling that allows the graphical definition and the validation of the models, referred to thereafter as ”ROS tooling”.

The ROS tooling is being developed, improved and tested for real use cases by the German funded project SeRoNet [24]. For this project different institutions (from research centers to robotics OEMs companies) joined their efforts to create a platform that unifies the development of robotics software supporting different middlewares like OPC-UA [25], ROS or SmartSoft.

A. *ROS metamodel*

The ecore ROS metamodel describes the concepts of the main three ROS dimensions:

- the filesystem (how code is organized and stored);
- the computation graph (how systems are split in processes and how these interact);
- the deployment mechanism (how entities are distributed, named, accessed at runtime).

By layering such a minimal model on manually developed ROS nodes and their disk location (the ROS package container) we can model also the “simple plumbing” infrastructure, i.e., the interaction patterns, by describing the communication interfaces, and set the basis for deployment phase, that is, artifact names. The ROS metamodel defines the interfaces of the communication by the type of communication (one-to-one, many-to-many or state machine pattern); the direction of the information (input or output); and by the communication object (the data structure of the messages being exchanged: message, service or action).

B. *Component Interface metamodel*

The component interface aims to achieve two main goals: 1) to simplify the deployment process of ROS systems with the concept of composition of sub-systems, and 2) to facilitate the creation of hybrid systems (in terms of interoperability with other frameworks, ideally component-based). Given these goals, and inspired by the Object Management Group (OMG) specification “Deployment and Configuration of Component-based Distributed Applications” [26], the previous ROS metamodel (Sec. III-A) was transformed to this generic “standard” concept. That is, according to the OMG , “a named set of provided and required interfaces that characterize the behavior of a component”. To group entities (“components”, in the parlance of this subsection) ROS provides a hierarchical naming structure used for all resources in a ROS computation graph, where each resource (node, topic, etc) is defined within a namespace. To preserve the nature of the original ROS code, the Component Interface metamodel refers to the topics, services and actions definitions from the ROS specific model and only adds the definition of namespaces for the entire component and/or for each interface.

C. *System metamodel*

The third metamodeling tool of this family makes use of Component Interface models to compose ROS nodes, subsystems, and systems. Such composition is achieved in ROS through the use of launch files, in which the integrator defines the nodes to be started, the package containing each of them, the arguments to be parsed and their grouping in namespaces and/or machines within which the nodes will be started. Another embedded tag available in launch files is the remapping one, used to define complex name assignments. This system, which currently have to be written manually, can only be validated at runtime. The ROS system metamodel was created to possibly validate at design-time the respective interconnections between nodes.

For its implementation the OMG specification “Deployment and Configuration of Component-based Distributed Applications” [26] was also taken into consideration. Such specification defines a connection as “either a communication path among the ports of two or more subcomponents allowing them to communicate with each other, or (it is) a communication path between an assembly’s external ports and an assembly’s subcomponents that delegates the external ports behavior to the subcomponents ports”. For the ROS case, this resulted in *TopicConnections*, *ServiceConnections* and *ActionConnections*. Using the tooling it is possible to validate the composability of nodes and identify the disparity of a communication object, i.e., the subscriber of a topic asking for a different message type than the one being published. The identification of such disparity results in an error emitted at design-time. Once the interconnection of different ROS nodes is validated, the tooling generates automatically the deployment artifacts code (i.e. *roslaunch*). This avoids the debug at runtime of errors often simply due to typing mistakes, and ensures the successful connections between the defined components.

IV. AUTOMATED MODEL GENERATION FROM ROS SOFTWARE ARTIFACTS

Our work aims to build upon the related work in order to:

- Leverage the many existing, manually written ROS components and the familiarity with ROS conventions among robotics software practitioners
- Leverage model-based techniques to improve the understanding of the existing code through automatically extracted models. Models that make use of our previous work (Sec. III) can be composed, with correctness and verification checks performed before deployment
- Systematize the definition and adoption of best practices by examination of such models, specifically with regards to interaction patterns

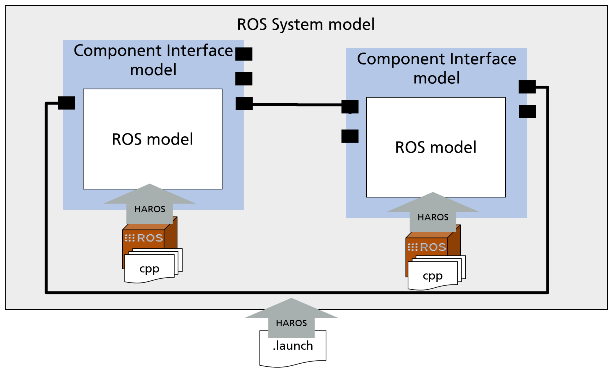


Fig. 1. Architecture overview of the approach using HAROS to extract the models

Given its open source nature and its federated development process, the sharing culture within the ROS community encouraged modularization of the software in order to facilitate the interoperability with software artifacts developed by others. In order to fit the needs of this characteristic community, we focus our efforts, at a first instance, on automatically generating the models by analyzing existing code. As a second strategy, also to improve the understanding of large ROS systems, whose source code is not necessarily always available, we complement this with a runtime monitor, which automatically extracts the ROS system model (Sec. III-C) of the running system it is started with.

A. Method 1: Static Code Analysis

For the code analysis we use the framework HAROS [27]. Supporting an extensive list of options, HAROS can be configured to check custom error types. For our concrete case, the analysis of ROS C++ code extracts a simplified (but expressive enough for our purpose) syntax tree of the programs.

For the purposes of this work, we created a plug-in that builds a model (conforming with the ROS metamodel in Sec. III-A) out of the simplified syntax tree of the ROS C++ code that the parsing and extrcation functions of HAROS yield. Concretely, it extracts the name of the package that contains the node, the name of the artifact that runs the node, the name of the node itself, and the list of the communication interfaces of the node (topics, services and actions) with other nodes in the system, respectively annotated with message types. HAROS actually extracts a considerably larger amount of information, which however is not needed and thus not being processed by our plug-in. The mentioned plug-in is publicly available and can be used for the generation of two different types of metamodels (Sec. III):

- **Model extraction of a single node:** the input for this scanner is the name of a node and the ROS package that contains it. The automatically generated result is a ROS model (Sec. III-A) for a node in Xtext grammar. By importing it into the ROS tooling, it can be visualized and integrated with other nodes.
- **Model extraction of a system and its constituent components:** the ROS nodes can be grouped so as to be started together using launch files. This extractor option takes as input the name of a launch file and the name of the ROS package that includes it. Thanks to HAROS and its launch files parser we can detect the names of the packages and names of nodes whose models compose the full system. Having obtained this list, we recursively call the extraction of models of single nodes (previous bullet point) to obtain all the required models of dependency packages. The parser is also able to detect the namespace where each node is started (instances of the source code classes), in our case defined in the ComponentInterface metamodel (Sec. III-B) within a system. To complete the analysis and give to the user the full package, together with the the models of the nodes we also automatically generate the rossystem model (Sec. III-C). Fig. 1 shows a schematic diagram of this approach.

While using HAROS, we are also aware of some of its limitations. Specifically, 1) it does not support ROS actions [28], and 2) it can only be used to analyze C++ code, while ROS is multilingual and supports also Python, Java, Lisp and Lua, among the others. Part of the technical contribution of this publication is the improvement of HAROS for a minimal support of ROS actions.

B. Method 2: Runtime Systems Monitoring

With static analysis being the most convenient approach to extract models from source code, we pursued the goal of automated generation of models from ROS artifacts also for already deployed, running systems. The idea is to monitor a running ROS system and inspect the communication between all its executing programs in order to extract the nodes, components, and system models, to then import them into the metamodels ecosystem of the ROS tooling. With this complementary approach we cover the following use cases:

- Non-Open Source Software (OSS). Although ROS code is often OSS, this framework is also used for commercial applications where, due to business reasons or licensing matters, the source code is not available and the end user can only run the pre-compiled binary code.
- Unsupported ROS distributions. The model extractor was developed for the newest ROS distribution. However, since the first ROS release (March 2010), the framework evolved in different distributions (12 in total at the time of this writing), which analogously to the Linux kernel distributions are still being officially supported or are already marked as having reached ”end-of-life” (EOL). Structuring the communication interfaces of both components with a common model language facilitates the interoperability.

For this contribution we use a simple introspection method: we analyze all the running nodes and their calls through the ROS master (i.e., the central broker for communications between nodes).

This method is very useful to inspect and get an overview of a system, but from the qualitative point of view, it is not the most convenient analysis method. First, because this monitor can observe just a specific execution at a certain moment. This does not capture, for example, a very common programming strategy in ROS of a topic being subscribed within the call of a service client. And second, because one of the main motivations of our work is the analysis of software composition at design time, performed to avoid errors and thus problems at runtime.

Additionally, to further expand on this method there are other approaches under consideration, like creating a wrapper for the ROS master; or even to intervene at the transport level, by running a monitor in parallel during execution and intercepting any call to the master.

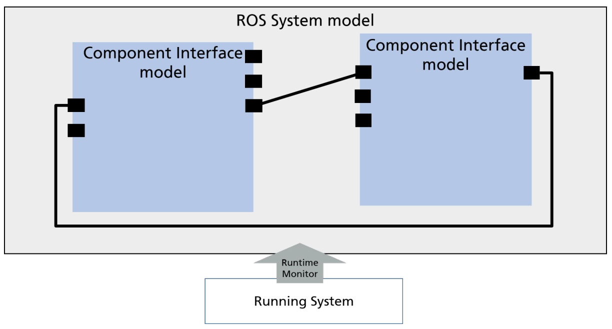


Fig. 2. Architecture overview of the approach using the runtime monitor to extract the models

As explained in Sec. III, the starting point of our architecture is a definition of a node based on two criteria: 1) the file system level, and 2) the computational level. Unfortunately, with the current implementation of the runtime monitor (using exclusively ROS framework sources) we can not obtain the first item, i.e., we are not able to extract the information about how the software is saved, distributed or organized on the disk. A future improvement of this approach combining the information we get from ROS (the current monitor) with the one that a Linux tool to manage processes provide (i.e. path to the executable file) could help to obtain further information and complete the model. In order to comply with the architecture, and to allow interoperability and full integration of models generated automatically through the runtime inspection method, the analysis at runtime generates two models. The first using the ROS metamodel (Sec. III-A) grammar (that generates a .ros file), and the second by defining the full system (Sec. III-C). See the Fig. 2 as overview. For the ROS model a fictitious ROS package is defined to hold all the nodes and their interface descriptions. In this generated model we see one of the limitations of this approach: the runtime monitor cannot differentiate the information of two of the ROS dimensions: the deployment and the filesystem level. Concretely, it cannot disambiguate whether a node was launched on a specific namespace, or whether the original source code already identified the interfaces within a particular namespace infrastructure.

C. *Comparison of the two methods*

Both methods are conceived with the same purpose in mind: to auto-generate models. However, the concepts employed are very different. The static analysis of code allows early detection of errors, while the runtime monitor cannot prevent execution issues. It can however give the user a good understanding of the (current) behavior of a system. Structuring this information as a model, and thanks to the ROS tooling we provide, with the results from both methods the use of common specification and name conventions can be checked and the components, subsystems and systems composed and their interoperability (potentially even with other frameworks through model to model (M2M) transformations) verified at design time.

We can expect a higher rate of components and interfaces found by the runtime monitor analysis, mainly because of the following factors:

- HAROS does not support Python, one of the most used languages in ROS.
- The support of ROS actions for HAROS is still at prototypical stage, and as such it does not cover all the possible ways to define an action with ROS C++ code.

Conversely, runtime analysis presents the following issues:

- It does not support ROS service clients, since it is not possible to find them through the analysis mechanism which we currently adopt (based on rosgraph introspection).
- It cannot extract filesystem information (part of this information is mixed with the one related to deployment on the resulting models), which makes it unfeasible to use this method to detect common patterns of interaction to then systematize their adoption (feature explained in the next Sec. V-B).

In Sec. VI we quantitatively compare a real example of results obtained by both approaches.

V. Leveraging Models in Manual ROS Code Development

With the tools for the auto-generation of models implemented and integrated successfully with the metamodels and ROS tooling, our next step is to exploit these tools in two phases. First, by providing a cloud system for convenient and, potentially, large-scale analysis of ROS code (given a quick enough uptake from the ROS community), and second by collecting generated models in a database so as to allow for comparisons and, ultimately, systematization of the adoption of common patterns and practices.

A. *Cloud Tools for Automated Model Generation*

We use the metamodels defined with a platform independent language and with accompanying tooling developed in Java within an Eclipse environment. In addition, to generate the models a local installation of HAROS and its libraries and ROS packages is required. Seeing this as a limitation and entry barrier for some users, we decided to provide a cloud solution that, through a web interface, can be used to generate models from code publicly available (hosted on Git, the most commonly used platform to share code for the ROS community). The web interface runs a Docker container [29] with a pre-configured image, that already contains all the required Linux libraries, ROS packages and HAROS. This image also sets up and prepares the workspace and the extractor scripts. This Docker-based architecture allows also to trigger multiple jobs in parallel on several Docker containers. This feature (supported by the web interface) was a strong motivator for the cloud solution. By allowing large scale analysis of packages, this concept can be used to extract common patterns over large codebases, to then advise users about "de-facto" standards, and produce a good base set of re-usable models for the ROS tooling.

B. *Extracting Best Practices from Models*

The advantages of ROS in terms of fast-prototyping and federated development were clearly the factors promoting its wide expansion, but at the same time these characteristics spurred its growth without a common definition of specifications and interfaces. The increasing adoption of ROS in professional domains calls for the definition of conventions and best practices as statements to assure, on one hand, a minimum threshold of quality for the software and, on the other hand, the integrability and interoperability of the different modules and systems. However, there exists no tool or even common documentation (beyond what users spontaneously share over a wiki) collecting all this knowledge in a formal format. Neither exists a quantitative global study analyzing the use of the different patterns to define a proper set of best practices. With the structures to formalize ROS interfaces in place (Sec. III-A), a tool for automatic large scale analysis available (Sec. V-A), and and awareness of the need for common specifications, we put efforts toward an extra technical contribution to analyze a diverse set of drivers for different devices types to create a common set of specifications. This is performed in two separate steps:

1) Detection of commonly used communication objects (the messages types of the communications between nodes) and their provision as a basic dictionary. This dictionary is publicly available and will be automatically loaded to any new ROS project created within the ROS tooling.
2) Detection of common patterns. In collaboration with the EU H2020 project ScalABLE 4.0 [30] we created a growing database of specifications (patterns for typically used robotics components, e.g., actuator controller, sensors, I/O devices...) by analyzing systematically the models of diverse drivers for types of devices. For this concrete project's use case the component specifications aim to help the configuration and use of a task orchestrator.

To complement this set of models, we contributed a new wizard for the ROS tooling to compare one-by-one the interfaces of two models (the target one, and a standard one or a custom specification). This returns a diagnostic of the comparison, listing as errors the cases that will interfere on the integration of the component with other generic standardized modules, like:

- the use of non-common messages (communication objects). This issue produces a mismatch between both sides of the communication channel.
- the absence of a required interface. This issue prevents the establishment of the communication.

The experiment section contains an example demonstration of this feature (Sec. VI-E)

VI. Use case example

All the tools and concepts presented in this paper are being improved, tested and evaluated on real demonstrators in the context of the ongoing projects SeRoNet and ScalABLE 4.0. In order to practically demonstrate the concrete technical contributions for this paper, we report in this section a use case based on the commercially deployed service robot Care-O-bot-4 (Fig. 3). Given its complexity, we can consider it an adequately representative example for service robotics applications.

Fig. 3. Care-O-bot4 full robot. Developed by Mojin Robotics.

The goal of this section is both to highlight the advantages and convenience of our tools when applied to the typical tasks that a robotics software engineer performs, and to point out the limitations of the different approaches, to be possibly tackled as future work. So to facilitate the understanding of the next subsections and of the vocabulary used to described the experiments, we first list the set of publicly available repositories from which to source the companion material:

- ***ROS Tooling***: this repository contains the ROS tooling infrastructure and also serves as the storage for documentation. By tooling we mean the Eclipse environment and the full Java and ecore implementation of the metamodels, as well as the Xtext and Xtend grammar implementations, a set of wizards for the graphical representations, and the support tools to automatically import, create or modify the models. ⇒https://github.com/ipa320/ros-model
- ***ROS Cloud tool***: this repository contains the backend code of the web interface publicly available to extract models (i.e. http://153.97.4.193/). We made this repository public for the cases where the source code is not hosted online. Furthermore, all the script tools used

to extract models are available within this repository. ⇒https://github.com/ipa320/ros-model-cloud

- ***ROS Graph monitor***: This repository holds the ROS package used for the monitoring and extraction of models at runtime. ⇒https://github.com/ipa-led/ros_graph_parser
- ***ROS Experiments***: this repository contains the results of different analysis. Including the full results of the experiment explained in this section correspond to the version cob4-25. ⇒https://github.com/ipa-nhg/ros-model-experiments

To perform the system experiments, we used the software description of the drivers that manage the hardware of the cob4-25, consisting in a total of 38 software components. We selected the most characteristic modules of which a real robot consists of (for the mechanic and sensing point of view), and provide interfaces to run a complete application. Listing some of them, we have included for instance the base with three wheels; the joystick to teleoperate it; three 2D laser scanners used to navigate the environment; four 3D cameras for visualization tasks; components like light, mimic control, and sound for general robot-user interaction. Fig. 3 shows the real aspect of one of the robot of the series.

A. ROS model extraction through static code analysis

The first step to import models into the ROS tooling is to statically analyze the atomic, self-contained software entities in ROS, that is, the nodes. To perform this step, we invoked the HAROS framework providing as input the name of the package that contains the C++ code and the name of the node. To cover all the possible cases, we provide to the user different methods: the cloud web interface, a locally running extraction script (this requires a local installation of HAROS), and the provided pre-configured Docker container.

For this concrete walk-through we opted for the cloud solution, providing as input (the information of one of the scanners node):

- *Git repository* https://github.com/ipa320/cob_driver
- *Package* cob_sick_s300
- *Node name* cob_sick_s300

The code in Lst. 1 shows the result of the analysis. Importing the auto-generated model to the ROS tooling we can visualize the driver of this scanner as showed in Fig. 4

```
PackageSet { package {
  CatkinPackage cob_sick_s300 { artifact {
    Artifact cob_sick_s300 {
      node Node { name cob_sick_s300
        publisher {
          Publisher { name 'scan'
            message 'sensor_msgs.LaserScan'},
          Publisher { name 'scan_standby'
             message 'std_msgs.Bool'},
          Publisher { name '/diagnostics'
            message 'diagnostic_msgs.DiagnosticArray'}}}}}}}}
```

Listing 1. sick_s300.ros file in Xtext format generated automatically by the cloud tool

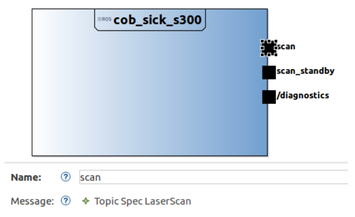


Fig. 4. Tooling visualization of the auto-generated model for a Sick s300 scanner driver

B. System model extraction through static code analysis

The first section analyzed only one of the nodes that compose the full robot. Performing such an operation independently for every single node is a tedious task. To facilitate the analysis of a system composition we use the launch file parser of HAROS. Practically speaking, launch files are written in XML format and are used in ROS to start together several programs.

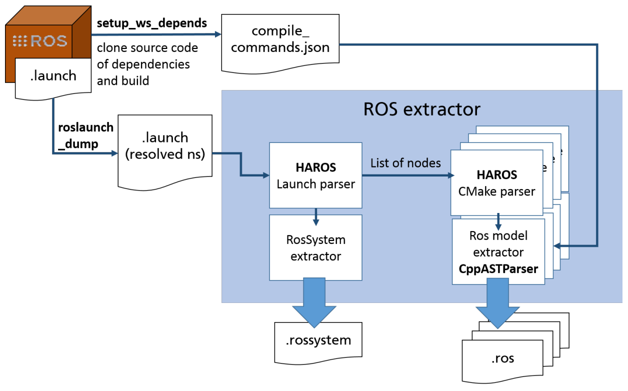


Fig. 5. Diagram of the approach of system model extraction through static code analysis

Fig. 5 shows an overview of the approach. First of all, we have to locate the source code of all the ROS software required to run our system. Typically, ROS system integrators install a released stable version of the dependencies in binary form, but for this analysis we need a local build of the full source code and all its dependencies. To automate the solution of this issue, we created a script that first asks ROS for all the dependencies of the target package (command *rospack depends-indent*) and obtains for each single package the GitHub URL that holds the source code (command *roslocate info*). Once we have this information, we download all the source code to our workspace and compile it, with the isolated option, all these packages together (*catkin_make_isolated*).When the build completes, we combine all the isolated builds, making our workspace ready to be analyzed with HAROS. The launch files in ROS are very powerful and can be used to manage and potentially fully redefine the behavior of a robot. This also means that they are very complex and hard to be fully supported by any automated tooling. Even more so when we consider a Care-O-bot whose launch file architecture is defined to support all the possible combinations of modules and configurations. Ultimately, this results in the main launch file being translated into an entangled combination of launch and XML files, with recursive inclusions and parsing of arguments and parameters. To streamline this analysis we created a script (*ros-model-experiments/tools/roslaunch-dump*) that, given the original root launch file, is able to create a new one that resolves all the namespaces (ns), names of the parameters and the monolithic includes of single nodes and their parameters.

Having this obtained launch file, the next step is to use the HAROS launch parser to detect the name of the packages and name of nodes, and the namespace that organizes the nodes. With this information we invoked the extractor (same code than in Sec. VI-A but called recursively) to generate automatically:

- a set of **.ros** files (including the model of the Sick S300, i.e. Lst. 1) ⇒*ros-model-experiments/cob4-25/cob4-25_static/rosnodes*
- a rossystem file that contains all the components with their list of remapped interfaces and its references to the original ROS model. A short part of this file is shown in Lst. 2 ⇒*ros-model-experiments/cob4-25/cob4-25_static/cob4-25_experiment.rossystem*

```
RosSystem { Name cob4−25
  RosComponents ( ....
    ComponentInterface { name '/base_laser_right/driver'
         NameSpace '/base_laser_right/'
      RosPublishers{
        RosPublisher '/base_laser_right/scan_raw' { RefPublisher '
            cob_sick_s300.cob_sick_s300.cob_sick_s300.scan'},
        RosPublisher '/base_laser_right/scan_standby' {
            RefPublisher 'cob_sick_s300.cob_sick_s300.
            cob_sick_s300.scan_standby'},
        RosPublisher '/diagnostics' { RefPublisher 'cob_sick_s300
            .cob_sick_s300.cob_sick_s300./diagnostics'}}....}}}
```

Listing 2. Sample of the ROS system metamodel generated by the static code analyzer

Examining the quality of the result, we found three concrete launch file structure limitations:

- The name of the interfaces is parsed as input argument. An example of this issue is the node *scan_unifier_node* of the package *cob_scan_unifier*. To solve this problem, we made a small modification to the original source code to obtain statically the name of the interfaces. However, this case should be supported and the code not modified.
- The type and name of the interfaces are parsed as parameters. This issue is made evident by a very concrete and common ROS framework: *ros_control*. This software can be called with a list of arguments that set the type of the controller to be started (i.e., joint_trajectory_controller, velocity_controller, position_controller..), and with this argument the controller manager checks through calls to the ROS parameter server at runtime the configuration and name of the interfaces providing such command. This case is quite complex to be supported by a static C++ code analyzer. To momentarily solve this issue, also knowing that once the type of controller is defined the delivered interfaces are fixed, we defined a special plug-in for our extractor that checks the arguments of the *controller_manager* node using a pre-defined template for the *ros_control* model

For a complete quantitative analysis of the result the next step of our experiment is the introspection of the system at runtime, and the comparison of results obtained by both methods.

C. System model extraction through runtime monitor

For this part of the experiment, we used the hardware of the real robot (Fig. 3) and started all the components used for the previous section analysis, in order to obtain comparable results.

Starting the same launch file (*ros-model-experiments/cob4-25/cob4-25_experiment.launch*)and initializing the actuators of the base (the wheels) to make their commands available, we called our ROS graph monitor (Sec. IV-B) to extract the model of the system. The result of this experiment are two files:

- the fictitious nodes model. A ROS file model (Sec. III-A) with a single ROS package containing all the nodes running on the system and the interfaces that these nodes offer to the rest of the network ⇒*ros-model-experiments/cob4-25/cob4-25_runtime/rosnodes/dump.ros*.
- a model of the system. A ROS system file model (Sec. III-C) containing a component definition for all the previously found nodes and referencing all the interfaces of the ROS model file. A short excerpt of this file is shown in Lst. 3 ⇒*ros-model-experiments/cob4-25/cob4-25_runtime/cob4-25.rossystem*

```
RosSystem { Name 'cob4−25'
  RosComponents (....
    ComponentInterface { name '/base_laser_right/driver'
      RosPublishers {
        RosPublisher '/base_laser_right/scan_standby' {
            RefPublisher 'dump_pkg./base_laser_right/driver./
            base_laser_right/driver./base_laser_right/scan_standby
            '},
        RosPublisher '/base_laser_right/scan_raw' {RefPublisher '
            dump_pkg./base_laser_right/driver./base_laser_right/
            driver./base_laser_right/scan_raw'},
        RosPublisher '/diagnostics' {RefPublisher 'dump_pkg./
            base_laser_right/driver./base_laser_right/driver./
            diagnostics'}}....}}}
```

Listing 3. Sample of the ROS system metamodel generated by the runtime monitor

D. Comparison of system extraction through static code analysis and runtime monitoring

Table I shows the quantitative comparison of both approaches in terms of number of components and interfaces found.

Unsurprisingly, the ROS Graph monitor is able to find all the nodes. This is expected because ROS starts all the nodes included in a launch file. For the static code analysis we obtained 25 components (a total rate of a 65,7%). A

	Static Analysis	Runtime Analysis
Components	25	38
Topic Publisher	148	245
Topic Subscriber	23	54
Service Server	35	92
Service Client	2	0
Action Client	5	3
Action Server	4	6
Total interfaces	217	400

TABLE I
COMPARISON OF THE EXPERIMENTS VI-B AND VI-C

similar result is obtained by comparing the number of detected interfaces through the runtime monitor (400) and through the static code analyzer (217). However, taking into account the lack of Python support (with many of the tools and monitors on Care-O-bot being written in Python), we consider the obtained data a more than satisfactory result. This consolidates our belief about the static analysis approach as the right one to support the auto generation of models, and its preferred adoption to build a database of common patterns. This also further directs our next efforts towards the improvement of HAROS to mitigate the mentioned (Sec. IV-C) limitations.

We now consider usability and re-usability. The models extracted through the static code analysis (e.g. Lst.1) are completely truthful and the ROS metamodels structures fully fulfilled. If we compare this model with the corresponding "node" auto-generated by the runtime monitor, we find the following two types of information as missing or wrong:

- we are not getting the filesystem information (i.e., the name of the ROS package that contains this node). Without this information the ROS tooling is not able to auto-generate a valid launch file for the composition of this component with others.
- the information related to the name of the node is wrong. When the runtime model generator runs, the nodes and its interfaces are already remapped. This assignment of a namespace is done during the deployment phase (i.e. for the family of metamodels, included in the ROS system, see Lst. 2).

These issues, mainly the lack of the filesystem information, make it hard (almost useless) the reuse of the single extracted models from the ROS Graph monitor for the composition of new systems. The only aspect where the analysis of this file is helpful is to find commonly used communication object types (i.e., messages types) that upgrade the provided dictionary 1.

E. Replacement of a component using code scanning and comparison of specifications

To show the potential of large scale code analysis as made possible by our tooling, we chose a typical use case in robotics: a system integrator in need of replacing one of the components of a system (often because a component is not commercially available anymore). The process of replacing a robot component starts with an evaluation of the alternatives in the market, 1) in terms of hardware (physical dimensions and electronic requirements), and 2) in terms of compatible software. In our case, we will focus on 2D laser scanners (specifically, those for which a ROS driver exists) which can potentially replace the SICK S300 laser on the current setup of the Care-O-bot. The ROS wiki provides an open catalog of supported sensors [31]. From this list we filter those that use the same driver (usually from the same vendor, but for a different series), and those whose software is not up-to-date or not publicly available. We obtain a final list of seven scanners to be evaluated. In order to know which of them are compatible with the rest of our system, we first obtain the model for each of them using our cloud tooling for the extraction (i.e., http://153.97.4.193/) with the following input list:

HLS-LFCD LDS:

- *Git* https://github.com/robotis-git/hls_lfcd_lds_driver
- *Package* hls_lfcd_lds_driver
- *Node name* hlds_laser_publisher

Hokuyo:

- *Git* https://github.com/ros-drivers/hokuyo_node
- *Package* hokuyo_node
- *Node name* hokuyo_node

Pepperl Fuchs r2000:

- *Git* https://github.com/dillenberger/pepperl_fuchs
- *Package* pepperl_fuchs_r2000
- *Node name* r2000_node

Rplidar:

- *Git* https://github.com/Slamtec/rplidar_ros
- *Package* rplidar_ros
- *Node name* rplidarNode

Sick Safety Scanners:

- *Git* https://github.com/SICKAG/sick_safetyscanners
- *Package* sick_safetyscanners
- *Node name* sick_safetyscanners_node

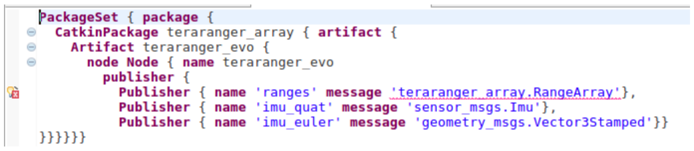
Teraranger Evo:

- *Git* https://github.com/Terabee/teraranger_array/
- *Package* teraranger_array
- *Node name* teraranger_evo

Neato XV-11:

- *Git* https://github.com/rohbotics/xv_11_laser_driver
- *Package* xv_11_laser_driver
- *Node name* neato_laser_publisher

Having obtained the auto-generated models (*ros-model-experiments/scanner_comparison*), we imported all of them into the tooling.



```
PackageSet { package {
  CatkinPackage teraranger_array { artifact {
    Artifact teraranger_evo {
      node Node { name teraranger_evo
        publisher {
          Publisher { name 'ranges' message 'teraranger_array.RangeArray'},
          Publisher { name 'imu_quat' message 'sensor_msgs.Imu'},
          Publisher { name 'imu_euler' message 'geometry_msgs.Vector3Stamped'}}
}}}}}}
```

Fig. 6. Teraranger Evo auto-generated model imported on the tooling

The **first** check is the evaluation of the use of standard messages. This check was not passed by the *Teraranger Evo*(as shown in the Fig. 6) and by the *Sick Safety Scanners* because both use non-generic message types (i.e., the use custom-defined ones) for the communication. The use of communication objects not included in the ROS tooling dictionary can be solved by updating it (one of the tools that we provide calls a script to import automatically new communication objects), but this is only recommended after a systematic evaluation of existing software and matched patterns.

```
Validate the file: hlds_laser_publisher.ros
for the specifications model: Laser2DScan.ros
OK:
- OK: Publisher for message type sensor_msgs/
    LaserScan found: scan -> scan
```

Listing 4. result of the comparison of the hlds_laser_publisher with the specification Laser2DScan generated by the tooling

In the case of the scanners, through a previous analysis of several nodes we already distilled a de-facto "standard" specification model from commonly used code (*ros-model-experiments/scan_comparison/Spec/Laser2DScan.ros*). The **second** part of this test is to compare all the automatically obtained models with this generic specification by using the comparison models tool V-B. All the analyzed package except the *Teraranger Evo* passed this check, with the diagnostic tool providing the output showed in Lst. 4. This means that they fulfill the requirement of publishing their output through the standard message type *sensor_msgs/LaserScan*. Lst. 5 shows the result of an unsuccessful check.

```
Validate the file: teraranger_evo.ros
for the specifications model: Laser2DScan.ros
ERRORS:
- ERROR: missed a publisher for message type:
    sensor_msgs/LaserScan
```

Listing 5. result of the comparison of the teraranger_evo with the specification Laser2DScan generated by the tooling

The **third** part of this test is to compare the models obtained automatically with the model of the current scanner mounted on the Care-O-bot, the SICK S300. The model of this driver is shown in Lst. 1. Unsurprisingly, none of the nodes passed the test, whose result is shown in Lst. 6. As for Care-O-bot there is a special best practice requirement, that all the drivers report constantly the current status of the hardware by publishing a diagnostics message. The lack of the diagnostics message will produce a warning for the Care-O-bot. The other error pointed by the test is due to the SICK S300 having non-common standby mode which can be published by its driver. However, this property is not needed and its absence is not considered an error and neither a warning.

```
Validate the file: hokuyo_node.ros
for the specifications model: sick_s300.ros
ERRORS:
- ERROR: missed a publisher for message type:
std_msgs/Bool
- ERROR: missed a publisher for message type:
diagnostic_msgs/DiagnosticArray
OK:
- OK: Publisher for message type sensor_msgs/
    LaserScan found: scan -> scan
```

Listing 6. result of the comparison of the hokuyo_node with the sick_s300 model generated by the tooling

VII. Conclusion and Future Work

The work presented in this paper aims to improve the software quality of ROS systems, in a complementary fashion to other initiatives within the ROS community. The basic ideas of this approach are: the exploitation of information available at design time, but mostly not used due to insufficient tooling, in order to avoid errors at runtime such as naming or message format mismatches; the identification of de-facto standard practices whose reuse, empirically, often improves quality through better understanding of the developed code by other developers and easier reuse/replacement of components. We pursued this approach through model generation by: automatic model extraction, both from source code through static code analysis and from binaries through runtime monitoring; and by making available the necessary tooling both as source code and as a cloud service, together with the starting nucleus of a model database to be used and expanded by the ROS community. We hope that this initial effort can contribute forms of model-based verification of manually written ROS code and push towards a future standardization of ROS interfaces beyond the practice of manually annotating and inspecting wiki documentation. In addition to growing the models database and involving further ROS developers, we plan two further directions for our future work. The first is to merge our cloud-based service for model generation with the Eclipse tooling into a more comprehensive cloud-based IDE, necessitating no installation and offering "IntelliSense-style" suggestions to e.g., pick the most adequate message format given a comparison performed in the background between the extracted model and matches in the database. The second is to ease interoperability between other frameworks, such as those leveraging OPC-UA servers and clients, through M2M transformations methods. This would improve the applicability of ROS within the manufacturing domain and other contexts.

Acknowledgment

The research and development project that forms the basis for this report is partially funded within the scope of the "PAiCE Digitale Technologien für die Wirtschaft" technology programme run by the German Federal Ministry for Economic Affairs and Energy and is managed by the DLR project management agency "Gesellschaft, Innovation, Technologie - Informationstechnologien/Elektromobilität" at the German Aerospace Center in Cologne.
The HAROS contribution 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).
Other parts of this work have been supported by the European Commissions Horizon 2020 Programme under the grant agreements no. 723658 (SCALABLE) and no. 732287 (ROSIN). The author is responsible for the content of this publication.

REFERENCES

[1] N. Hammoudeh Garcia, M. Lüdtke, S. Kortik, B. Kahl, and M. Bordignon, "Bootstrapping mde development from ros manual code - part 1: Metamodeling," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, Feb 2019, pp. 329–336.

[2] M. Hägele, "Robots conquer the world [turning point]," *IEEE Robotics Automation Magazine*, vol. 23, no. 1, pp. 120–118, March 2016.

[3] "Industrie 4.0 an der Börse: Software lukrativer als Hardware." [Online]. Available: https://www.wiwo.de/finanzen/geldanlage/industrie-4-0-an-der-boerse-software-lukrativer-als-hardware/14452080-3.html

[4] "IEEE RAS Technical Committee for Software Engineering for Robotics and Automation." [Online]. Available: https://www.ieee-ras.org/software-engineering-for-robotics-and-automation

[5] "Robotics Software Engineer Nanodegree Program." [Online]. Available: https://eu.udacity.com/course/robotics-software-engineer--nd209

[6] L. Gherardi, D. Hunziker, and G. Mohanarajah, "A software product line approach for configuring cloud robotics applications," in *2014 IEEE 7th International Conference on Cloud Computing*, June 2014, pp. 745–752.

[7] D. Brugali and N. Hochgeschwender, "Software product line engineering for robotic perception systems," *International Journal of Semantic Computing*, vol. 12, pp. 89–107, 03 2018.

[8] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel, "The smartmdsd toolchain: An integrated mdsd workflow and integrated development environment (ide) for robotics software," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, pp. 3–19, 08 2016.

[9] B. Gerkey, K. Stoy, and R. T. Vaughan, "Player robot server," Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep., November 2000.

[10] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006, http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.

[11] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 87–98.

[12] 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.

[13] "ROS: Robot Operating System," http://www.ros.org/, accessed: 2019-04-25.

[14] L. Zhang, R. Merrifield, A. Deguet, and G.-Z. Yang, "Powering the world's robots—10 years of ros," *Science Robotics*, vol. 2, no. 11, 2017. [Online]. Available: http://robotics.sciencemag.org/content/2/11/eaar1868

[15] "ROS metrics wiki site." [Online]. Available: http://wiki.ros.org/Metrics

[16] R. Kittmann, T. Frhlich, J. Schfer, U. Reiser, F. Weihardt, and A. Haug, "Let me Introduce Myself: I am Care-O-bot 4, a Gentleman Robot," in *Mensch und Computer 2015 Proceedings*, S. Diefenbach, N. Henze, and M. Pielot, Eds. Berlin: De Gruyter Oldenbourg, 2015, pp. 223–232.

[17] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480693

[18] A. Bubeck, F. Weisshardt, and A. Verl, "BRIDE - A toolchain for framework-independent development of industrial service robot applications," in *ISR/Robotik 2014; 41st International Symposium on Robotics*, June 2014, pp. 1–6.

[19] P. Kilgo, E. Syriani, and M. Anderson, "A visual modeling language for rdis and ros nodes using atom3," in *Simulation, Modeling, and Programming for Autonomous Robots*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 125–136.

[20] "RobMoSys - Composable Models and Software." [Online]. Available: https://robmosys.eu/

[21] C. Schlegel and R. Worz, "The software framework SMARTSOFT for implementing sensorimotor systems," in *Proceedings of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 1999, pp. 1610–1616 vol.3.

[22] 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.

[23] A. Santos, A. Cunha, and N. Macedo, "Static-time extraction and analysis of the ROS computation graph," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, Feb 2019, pp. 62–69.

[24] "SeRoNet project website." [Online]. Available: https://www.seronet-projekt.de

[25] "Unified Architecture - OPC Foundation." [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/

[26] OMG, *Deployment and Configuration of Component-based Distributed Applications Specification Version 4.0*. Object Management Group, 01 2006.

[27] 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)*, Sept 2017, pp. 3855–3860.

[28] "ROS actionlib stack documentation," http://wiki.ros.org/actionlib/, accessed: 2019-04-25.

[29] "Docker project webside," https://www.docker.com/, accessed: 2019-04-25.

[30] "ScalABLE 4.0 project website." [Online]. Available: https://www.scalable40.eu/

[31] "Catalog of ROS supported sensors." [Online]. Available: http://wiki.ros.org/Sensors