Software and Systems Modeling (2021) 20:2047–2070
https://doi.org/10.1007/s10270-021-00873-2

SPECIAL SECTION PAPER

Bootstrapping MDE development from ROS manual code: Part 2—Model generation and leveraging models at runtime

Nadia Hammoudeh García[1] · Harshavardhan Deshpande[1] · André Santos[2] · Björn Kahl[1] · Mirko Bordignon[3]

Received: 1 March 2020 / Revised: 20 January 2021 / Accepted: 6 February 2021 / Published online: 19 April 2021

Abstract
Model-driven engineering (MDE) addresses central aspects of robotics software development. MDE could enable domain experts to leverage the expressiveness of models, while implementation details on different hardware platforms would be handled by automatic code generation. Today, despite strong MDE efforts in the robotics research community, most evidence points to manual code development being the norm. A possible reason for MDE not being accepted by robot software developers could be the wide range of applications and target platforms, which make all-encompassing MDE IDEs hard to develop and maintain. Therefore, we chose to leverage a large corpus of open-source software widely adopted by the robotics community to extract common structures and gain insight on how and where MDE can support the developers to work more efficiently. We pursue modeling as a complement, rather than imposing MDE as separate solution. Our previous work introduced metamodels to describe components, their interactions, and their resulting composition. In this paper, we present two methods based on metamodels for automated generation of models from manually written artifacts: (1) through static code analysis and (2) by monitoring the execution of a running system. For both methods, we present tools that leverage the potentials of our contributions, with a special focus on their application at runtime to observe and diagnose a real system during its execution. A comprehensive example is provided as a walk-through for robotics software practitioners.

Keywords ROS · Models · MDE · Robotics

1 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 [32], making stock hardware platforms repurposable by means of software. Furthermore, the wider digitization trend affects 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 [6]. Hence, robotics software engineering is a distinctly recognized field, with dedicated publishing venues, technical committees [5], and educational curricula [8]. 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. These include model-based techniques, such as model-based software product lines (SPLs applied to robotics software, e.g., [21,31]) and model-driven engineering integrated development environments (MDE IDEs, e.g., [45]). Meanwhile, more traditional approaches broadly falling under the category of software frameworks have also been proposed [19,30,39,44]. Among them, the Robot Operating System (ROS [10]) gained widespread

Communicated by Tao Yue, Man Zhang, and Silvia Abrahao.

✉ Nadia Hammoudeh García
nadia.hammoudeh.garcia@ipa.fraunhofer.de

Harshavardhan Deshpande
harshavardhan.deshpande@ipa.fraunhofer.de

André Santos
andre.f.santos@inesctec.pt

Björn Kahl
bjoern.kahl@ipa.fraunhofer.de

Mirko Bordignon
mirko.bordignon@ieee.org

1 Fraunhofer IPA, Institute for Manufacturing Engineering and Automation, Stuttgart, Germany

2 INESC TEC & Universidade do Minho, Braga, Portugal

3 Google Germany GmbH, Munich, Germany

acceptance in research and service robotics, with increasing signs of adoption in the industrial domain [47]. As developers of robot software using traditional frameworks for production-grade use cases, and as researchers in model-driven engineering approaches for robotics software, we identified possible ways to bring some advantages of model-based techniques into standard robotics software engineering practice. A user study documented in [45] reveals that "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 confirms this finding. Aiming at completely replacing manual code development with an all-encompassing MDE IDE has, from our point of view, historically proved difficult to impossible, given the large variety of application use cases and of target platforms to support. We thus instead aim at enabling developers to use model-based techniques as a complement, while leveraging large, preexisting codebases, such as the corpus of open-source ROS hand-written components (more than 3000 [9]), as base for new systems. 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. ROS is the de facto standard in robotics research, with a big user base [29] (pp. 5–6, 18–20, 24, -25). Even the International Federation of Robotics (IFR) in 2019 estimates its use to program service robotics applications in 70% of the cases.

ROS spreads quickly exactly due to the ease of use and minimal constraints, which means that it leaves ample room for errors and ambiguities. Some of them can be easily detected with a MDE approach, specifically: (1) many errors (e.g., message type mismatches) which are not detected until runtime; (2) proliferation of overlapping interfaces, whose consolidation is not enforced by the ROS architecture despite the many efforts within the community for their consolidation. For the ROS concrete case, but also a characteristic of large ecosystems, many developers are used to code manually, and therefore, much of the existing available code was manually written. In order to address this community, we focus our efforts, at a first instance, on automatically generating architectural models of the corresponding source code, via static analysis. 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 model of the running system it is started with.

After an analysis of the results and the use of the approach to analyze real use cases, we identified not only the shortcomings of our approach, but we also found an interesting and promising new application domain: the diagnosis of robotics systems at runtime. For this paper, in comparison with our previous publication [33], we introduce, as novel contribution, new features to mitigate partly or completely the limitations found for both previous efforts, i.e., (1) the techniques to extract models and (2) the metamodels definition, and also we show our first contributions to leverage our efforts to monitor large ROS systems during their execution.

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. Section 4 presents the basis of this paper: two approaches to automatically generate architectural models from existing ROS software artifacts, respectively, through static analysis of source code and runtime system monitoring. Section 5 describes our publicly available cloud infrastructure for extracting models from provided source code repositories, and a first database of models which was built from open-source ROS components.

To complement the description of the approaches, in Sect. 6, we compare the pros and contras of both extraction methods and present one of the main novel contributions of this paper, a tool to auto-combine the results obtained by both methods. This tool is based on lessons learned since our first experiments with model extractors [33]. Section 7 presents our second main new contribution: An adaptation of our metamodels to cover additional relevant runtime information required for performance analysis at system level and first experiments about how our models and infrastructure can be used to diagnose the performance of a running system during its execution.

Finally, Sect. 8 details, based on an use-case example, how our tooling can be used to extract models with the two approaches and evaluates and compares the results and their potentials. Using the use case, it also shows how to ease the replacement of sub-components by identifying common interaction patterns. The Care-O-bot 4 [37], a service robot now engineered and commercialized by the company Mojin Robotics, serves as 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. Section 9 recapitulates our contributions and lays out some directions for future work.

2 Related work

Firstly, we describe the work which we build upon, the Robot Operating System (ROS) and HAROS, a framework for static code analysis. Then, we evaluate the existing technologies related to our research: (1) model-driven engineering solutions to program robots and specially those which produce ROS code and (2) efforts to monitor and diagnose ROS.

2.1 ROS—the Robot Operating System

ROS [10,39] 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 and 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. Complementing these, a common ROS library offers an extra pattern, called action, that simulates a simple state-machine structure.

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.

In 2007, the first distribution of ROS was released, and in general, the robotics community changed significantly since then. In 2016, the ROS 2 project has been made public, with the main goal of adapting ROS to new needs such as improving quality of ROS code and security of ROS systems.

2.2 HAROS—static code analyzer framework for ROS

HAROS[1] is a plug-in-based framework whose primary focus is static analysis of ROS software [42]. 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 [40] (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 Python code and ROS launch files. 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 [40], 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. Thus, it enables both a wider range of analyses and a partial specification style, where the boundaries of analysis results can be progressively pushed. Metamodeling targeting ROS systems is separately examined in the next section.

2.3 MDE efforts on the robotics domain

Robotics systems can be seen as consisting of independent modules and the interfaces that allow their interaction. Based on this idea, Brugali et al. published in 2009–2010 [22, 23] a broad study about the application of component-based software engineering for robotics where the benefits of this approach in terms of re-usability were clearly identified.

The V3CMM-3 View Component Meta-Model [18], a platform independent modeling language, divides the development of robotics application into different views that cover the structural and the behavioral aspects of a component-based modeling language.

The main goals of this approach are the re-usability of models and facilitation of model transformations; the authors achieved them by the introduction of the standard Unified Modelling Language (UML) as base language.

More recent efforts applying MDE to robotics are: SmartSoft [43], a service-oriented component-based approach covering the entire robot development process from defining "services" (datatypes and their communication properties) and "components" (computation units that consume and produce data through "services") to designing full systems that are composed of those "components". The SmartSoft tooling generates the component's skeleton (a wrapper for the user code), including its interfaces and patterns for communication, based on service, component and system models that can be graphically edited. The Papyrus DSML RobotML [26], based on Component-Port-Connector (CPC) metamodel architectures, introduces a solution to design, simulate, and deploy robot applications.

The BRICS (Best Practices in Robotics) [24] 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 [25]) 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" [25], and showed how to fit ROS in a model-based approach. However, it main-

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

tained 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, for example, model-supported checking of component composition. Related to the aim of easing the interoperability of ROS with other systems, a visual modelling language was created to transform ROS-specific message definitions to Robot Device Interface Specification (RDIS) and vice versa, by mapping the communication mechanisms [36].

Estevez et al. [27,28] present a relevant example inline with our research to explore the advantages that model-driven engineering provides for the development of applications for the concrete case of robotic manipulators platforms. It provides a toolchain to go from Eclipse MDE models to ROS code for manipulation tasks.

The ongoing RobMoSys project [7] extends on SmartSoft [43] and promises predictable composition of both existing and new components through MDE.

2.4 Monitoring and diagnosing of ROS applications at runtime

The ROS diagnostics system is mainly designed to collect information from hardware drivers and robot hardware for analysis, troubleshooting and logging. The entire toolchain is composed of five components from interpreters of the collected information to graphical diagnostics visualizers. The architecture of this stack is quite simple; it collects all the information published in a specific message type called *diagnostic_msgs/ DiagnosticStatus* [3]. This information object contains the name of the device, its status and a short data message. The developer of a driver is responsible for adding the publisher with this diagnostic information when implementing the driver. Some ROS applications use this diagnostics tool not only for hardware components but also for software components including even monitoring programs. (An example related to our target experiment is the package cob_monitoring [2].) However, the main shortcoming of all these existing solutions is that they are specific for a concrete application and not very reusable.

The *model-based diagnosis and repair* [46] architecture tries to improve upon the limitations of the ROS diagnostics stack. It is an observer-based system that is meant to supervise the system running under ROS. An observer is a general software entity that monitors a particular aspect of the system. There are different kinds of observers depending on what aspect or property has to be supervised. The observations are processed by a diagnostics engine, and finally, the result is published to the global topic */diagnosis*. A rule engine is used to trigger events after processing observations or diagnoses. Unfortunately, the source code has a GPLv3 license, the repository link is broken and the project is unmaintained, discouraging us to use it for our scenario. Even other introspection tools [20] monitor the metadata of the communication between nodes and its interfaces in terms of utilized network bandwidth, message frequency, jitter and even latency. Though this is an important feature, this tool does not monitor the presence of required nodes or its interfaces in the running system. Another discouraging aspect of the tool is that it defines a custom ROS message type as an output of its diagnosis, which makes it difficult to plug-in to existing software components. However, in terms of reusability and integration, an optimal monitoring tool should be characterized by its flexibility by allowing interfacing with such external tools seamlessly.

3 Previous work: ROS metamodeling

Our previous works aimed to build upon the related work in order to:

- leverage the many existing ROS components and familiarity with ROS conventions among robotics software practitioners, rather than imposing, for example, a new MDE IDE;
- leverage model-based techniques to improve the understanding of manually written code through automatically extracted models, and enable automated correctness checks before deployment;
- systematize the definition and adoption of best practices by examination of such models, specifically with regard to interaction patterns (topics, services, messages): this activity currently depends largely on manual inspection of wiki documentation, in turn to be manually maintained.
- get a better understanding of the system architecture and facilitate its evaluation during the execution.

One of the main motivations of this work is to facilitate the composition of large systems, a task that ROS integrators perform on a regular basis because of the availability and reuse of off-the-shelf components. The lack of tools to validate, create deployment artifacts and test the composition at design time (currently a tedious trial-and-error process during the execution of the code) represents, from our point of view, one of the biggest shortcomings of ROS.

The contributions of this paper leverage an existing family of three metamodels which we developed in previous work [34]. These metamodels split the description of a 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 their 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 hereafter as "ROS tooling".

3.1 Metamodels

3.1.1 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, and 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. We can further extract artifact names as basis for describing the deployment phase.

The ROS metamodel defines the communication interfaces by three aspects: (1) the type of communication (one-to-one, many-to-many or state machine pattern), (2) the direction of the information (input or output), and (3) by the communication object (the data structure of the messages being exchanged: message, service or action).

3.1.2 Component interface metamodel

With the component interface metamodel, we aim to achieve two main goals: (1) to simplify the deployment process of ROS systems by leveraging 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* [38], the previous ROS metamodel (Sect. 3.1.1) 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 give to the reader an overview of our component interface metamodel and simplifying its implementation, the component interface represents all the ports (inputs and outputs) of a component or system describing them as communication interfaces, whose basic characteristics are the communication pattern (*how* communication is done) and the communication object (*what data* is communicated). To preserve the nature of the original ROS code, the component interface metamodel refers to the ROS interfaces (i.e., topics, services, and action definitions from the ROS specific model) and only adds the definition of namespaces for the entire component and or for each interface.

3.1.3 RosSystem 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 common use of launch files is to define complex name assignments. This allows to transparently remap resource lookups for name "A" with resource lookups (of the same type) for name "B". Furthermore, launch files can be (and are commonly) recursively included from other launch or XML files that configure the nodes by passing new arguments. This system, which currently has to be written manually, can only be validated at runtime. The RosSystem 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" [38] 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 port's behavior to the subcomponent's ports". For the ROS case, this resulted in a list of *TopicConnections*, *ServiceConnections* and *ActionConnections*. We designed for ROS systems a metamodel that apart of the definition of these connections, with their deployed name and the reference to the sender and receipt port, also includes the component interface metamodel. Figure 1 shows the class diagram of the RosSystem metamodel.

This simple definition fits perfectly to the ROS architecture and its deployment mechanisms and is powerful enough to allow the validation of the composability of nodes and the identification of any 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 by our

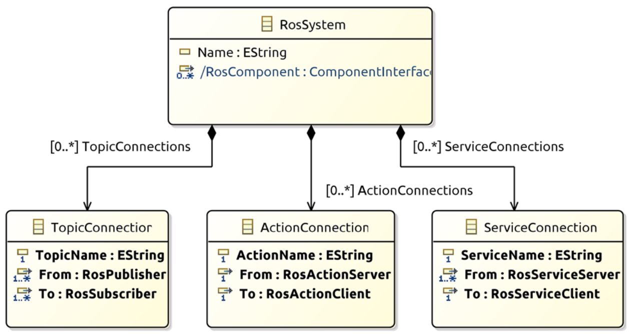


Fig. 1 Class diagram of the RosSystem metamodel

ROS tooling. Without the tooling, such type checks are only performed at runtime. There is no native design-time support in ROS. Moreover, once the interconnection of different ROS nodes is validated, the tooling generates automatically the deployment artifacts code (i.e., the *roslaunch* file and an installation script). This avoids the need to debug errors at runtime that are often caused by simply typing mistakes and ensures the successful connections between the defined components.

3.2 ROS tooling

The ROS tooling is being developed, improved and tested for real use cases by the German-funded research project SeRoNet [14]. For this project, different institutions (from research centers to robotics OEM companies) joined their efforts to create a platform that unifies the development of robotics software supporting different middlewares like OPC-UA [16], ROS or SmartSoft. We constantly integrate new applications, as a collection of plug-ins, into the SeRoNet tooling. These applications build upon our set of metamodels and the corresponding DSLs. These demonstrate the benefits of the combination of ROS and MDE techniques [35]. Among others benefits, we can mention:

- Diffuse ROS best practices. Promoting (with auto complete functions) the use of common specification patterns and the use of ROS naming conventions.
- Validation of system connections at design time. The grammar of our DSL for the definition of systems incorporates rules (for disparity of communication objects or agents) to evaluate the composition of systems.
- Auto-generation of valid code; from the definition of a single node to the composition of subsystems
- Introspection at design-time to improve the understanding of what will happen at runtime; graphical tools can simulate the runtime performance of the system without executing it.
- Interoperability with other frameworks; through model-to-model techniques we demonstrate the auto-generation of bridges from ROS to other frameworks.

4 Automated model generation from ROS software artifacts

4.1 Method 1: static code analysis

For the code analysis, we use the framework HAROS [41]. This powerful tool was designed with the main objective of early detection of problems during the software development phase, to give the user a diagnosis of the quality of the code. Supporting an extensive list of options, HAROS can be configured to check custom error types. For our concrete case, the analysis of the ROS code, it integrates the Bonsai [15] interface that 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 Sect. 3.1.1) out of the simplified syntax tree of the ROS C++ and Python code that the parsing and extraction functions of HAROS yield. 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) connecting it to 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[2] and can be used for the generation of two different types of models (Sect. 3):

- **Component model** This model is generated from a single ROS node, identified by a ROS package and the name of the node with in the package
- **RosSystem model** This model describes a complete (sub) system and all its constituents. It is generated from a single ROS launch file and the packages and files referenced from that launch file.

In the first case, generating a model representing a single ROS node, the scanner takes as input the name of a node and the ROS package that contains the node. The automatically generated result is a ROS model (Sect. 3.1.1) for a node in Xtext grammar. By importing it into the ROS tooling, it can be visualized and integrated with other nodes.

For the second case, generating a model representing a full RosSystem that consists of multiple nodes, the extractor

[2] https://github.com/ipa320/ros-model-cloud.

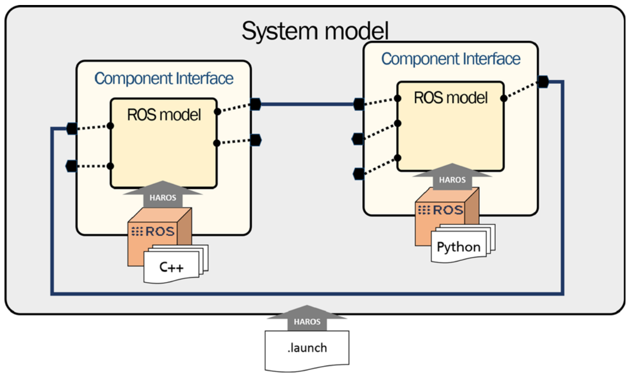


Fig. 2 Architecture overview of the approach using HAROS to extract the models. One parser analyzes a launch file to discover the system-level structure, i.e., RosSystem model and component interface models. A second parser extracts the ROS model from all discovered nodes

takes as input the name of a launch file that describes the system and the name of the ROS package that includes the launch file. The HAROS launch files parser can extract the names of the packages used in the system and the names of nodes that compose the full system. With the list of node names and packages, we recursively call the extraction of models of single nodes to obtain all the required models of dependency packages. The parser also extracts the namespace where each node is started (instances of the source code classes) within a system, which we need for the ComponentInterface metamodel (Sect. 3.1.2). The generated models of individual nodes and the extracted connectivity information are then combined into the full RosSystem model (Sect. 3.1.3). Figure 2 shows a schematic diagram of this approach.

Part of the technical contribution of this publication is the improvement of HAROS for the support of ROS actions.

HAROS, being an analysis framework that heavily relies on source code parsers and static analysis, comes with its own set of limitations. One of the most obvious limitations is language support: HAROS only supports C++ and Python, while ROS is a multilingual framework with client libraries for C++, Python, Lisp, Java, JavaScript, and more. Although a significant limitation, HAROS is sufficient for our use cases, considering that C++ and Python are the main client libraries used in the community, representing 85% of the available code base [9], while other bindings are more experimental.

Another significant limitation of HAROS is static analysis itself. As stated in Rice's theorem, any non-trivial semantic property of a program is undecidable. In this context, this may translate to ROS topic and service names that cannot be fully resolved, for instance, when their values are computed dynamically. This issue can be mitigated if developers follow coding guidelines—making the source code more predictable, easier to fit into a pattern. The ROS community has proposed some code quality guidelines, but these are largely ignored, resulting in a heterogeneous code base, despite some recurring patterns (Sect. 2.2). With the advent of ROS 2—which comes with its own set of guidelines and enforcing tools out of the box [12]—we expect better overall performance, precision and recall from model extraction based on static analyses. At the time of writing, support for ROS2 is still an ongoing effort for HAROS, which is able to handle ROS2 workspaces but lacks parsing support for the new client libraries.

4.2 Method 2: runtime systems monitoring

With static analysis being the most convenient approach to extract models from source code, we also pursued the goal of automated generation of models from ROS artifacts 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, the ROS 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 (13 in total at the time of this writing) with only the last 3 being maintained.

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).

As explained in Sect. 3, 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 model extractor (using exclusively ROS framework sources) we cannot 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 in addition to the system model an artificial (fictitious) instance of the ROS model for each

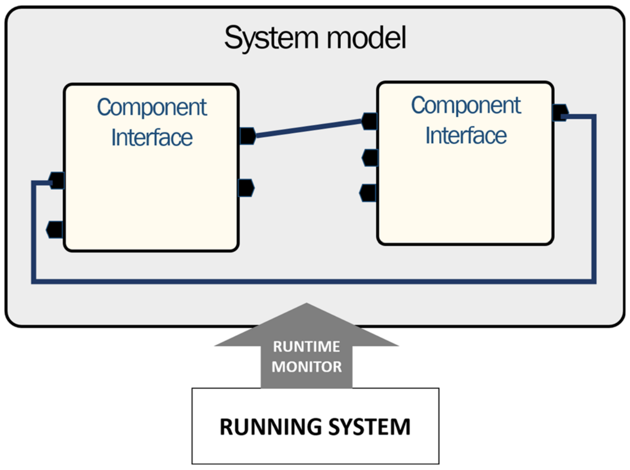


Fig. 3 Extracting models using the runtime model extractor. Only the system and component interface model can be deduced using ROS' introspection methods

node found by tracing the computation graph as available from the ROS master node. This allows to generate the full model as depicted in Fig. 2 although the runtime monitor can only "see" the system level including node interconnectivity, see Fig. 3. 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. More specifically, 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.

All the code related to the runtime model extractor is publicly available[3] and documented.

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 informative analysis method. First, because this monitor can observe only a specific execution at a certain point in time, therefore it does not capture, for example, a very common programming strategy in ROS, where a topic is being subscribed to within the call of a service client. Also analyzing an *already running* system does not match one of the main motivations of our work: The analysis of software composition *at design time* performed to avoid errors and thus problems at runtime. On the other hand, this method opens a new door for the analysis and diagnostics of systems at runtime based on modeling languages.

Later in this paper, in Sects. 6 and 7, we will show our efforts made to mitigate the drawbacks of this approach and to leverage its potential at runtime.

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

5 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. Firstly, 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 secondly, by collecting generated models in a database to allow for comparisons and, ultimately, systematization of the adoption of common patterns and practices. Both tools use the static code analysis method, since with the runtime introspection we 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.

5.1 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. The system can be accessed through a web interface to generate models from publicly available code (hosted on Git version-control systems, the most commonly used platform to share code for the ROS community). The web interface runs a Docker container [4] 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 script. We made also public[4] this docker image and the source code of the configuration for the analysis. This Docker-based architecture allows also the triggering of 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.

[3] https://github.com/ipa-nhg/ros_graph_parser.

[4] https://github.com/ipa320/ros-model-cloud.

5.2 Extracting best practices from models

The advantages of ROS in terms of fast-prototyping and federated development (such as simple txt-based message definition, robust, yet simple to use communication libraries, no dependence on specific IDEs) 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 highlights the need of conventions and best practices as statements to assure, on the one hand, a minimum threshold of software quality and, on the other hand, to allow the easy integrability 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. As can be expected, within the community there are common working groups (for applications like manipulation, task planning, navigation, etc.) where some common specifications are defined like *de-facto standards*, but without a fixed common format or specifications DSL. With the structures to formalize ROS interfaces in place (Sect. 3.1.1), a tool for automatic large-scale analysis available (Sect. 5.1) 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. Identification 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[5] and will be automatically loaded to any new ROS project created within the ROS tooling.
2. Identification of common patterns. In collaboration with the EU H2020 project ScalABLE 4.0 [13], 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.[6] 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 to 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 feature returns as a result a list of potential errors that may hinder the integration of the component with other generic standardized modules, like:

- the use of uncommon 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 (Sect. 8.1.2)

6 Evaluation of the extracted information and combination of both methods

The first part of the section evaluates the results obtained by both methods and their reliability when they are applied to real ROS robot systems; in other words, we aim to answer the following research question:

RQ1 *How complete is the information we can automatically obtain (from code-to-model extraction methods) to describe a large Robot Operating System application as a set of modular and reusable components and the interaction interfaces among them? Is there incorrect information?*

For our approach, this question is directly related to the following one: is the automatically extracted information enough to make a good use of our MDE ROS Tooling? Which provides us benefits like validation of the composition, autogeneration of deployment artifacts or interoperability with other frameworks.

Then, in the second part of this section, we introduce a new contribution. A feature that allows the combination of the outcomes obtained by both methods and, therefore, produces as result a complete model, in terms of expressiveness and amount of interfaces.

6.1 Comparison of static and runtime model extraction

We assume that our metamodels, defined from the three ROS pillars (i.e., (a) the filesystem, (b) the computation graph, and (c) the deployment information) and successfully used for the design and deployment of real systems are valid to describe a robotic application and secondly that the goal of the extractors is to instantiate them completely.

Both extractor 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 model extractor cannot prevent execution issues. It can, however,

[5] https://github.com/ipa320/RosCommonObjects.

[6] https://github.com/ScalABLE40/scalable_component_model.

give the user a good understanding of the (current) behavior of a system.

In [33], we analyzed the models auto-generated using both our methods (the static analysis of the code and the introspection at runtime of the application) on real robot systems. Considering the known limitations of both methods based on their design, the comparison can be shortly summarized by:

– With static code analysis, we are able to get all the information to completely instantiate all the fields of the ROS models (Sect. 3.1.1), which are the basis of our set of metamodels and, consequently, the basis of all the applications of our models in terms of validation, composability and interoperability. On the other hand, the static analysis performs worse out on the rate of interfaces found (detecting 10 to 50% less interfaces than the runtime monitor), mainly because of the dynamically assigned values on the code.
– With runtime monitoring, we are not able to completely instantiate the models (this method misses the filesystem information), and its biggest disadvantage is that in case a node failed and died we will not be able to detect it, which makes the models obtained via this method potentially unreliable. But, in comparison with the static analysis, we get a very large amount of data (up to the 90% of the interfaces running on the system).

The immediate conclusion of this analysis is that while one fails on the data volume, the other fails in expressiveness. Then, if we want to get the entire model as the most evident improvement we can perform to combine both approaches. Hereunder, we present a recent technical contribution that analyses and merges the result of both methods to obtain a more exhaustive model.

6.2 Solution: combination of the two methods

Considering the pros and contras and identifying the complementary advantages of each extraction method, we define different strategies (supporting different use cases) to appropriately combine the result of both methods. With this solution, we obtain a more precise model that the separate ones or a “simple” merge of the results that will produce a bunch of duplicated data. Before we identify the characteristics and potential uses of each strategy, we have to remark that, on the one hand, this feature will be used at design time but using the results of a previous analysis of the execution of a system. The runtime data can indifferently come from the real target system, from its simulated version or even from a run on a different machine of a set of the desired components. On the other hand, this solution considers only components, because it is the only reliable information we are able to extract from the runtime monitoring method. Also we assume an iterative design and development process, where we can subject early drafts of the to-be-developed system to runtime monitoring and integrate obtained insights into the further development process. This is a realistic scenario, since the federated development process and sharing culture in the ROS software ecosystem frequently have new developers build on preexisting (sub)systems.

We first define the distinct sets of component interface models we can obtain by comparing the extraction results of both extractors:

– Set $\mathbb{A}$ **Components from static and not in runtime analysis**: these are the components found during the static analysis but not present during the runtime extractions. It could contain components that were not started on our system or died before we recorded the data.
– Set $\mathbb{B}$ **Components from runtime and not in static analysis**: these are component found running on the system but not detected by the static analyzer, probably because the source code was not available for the analysis or not supported by HAROS. This includes component interfaces dynamically generated at runtime.
– Set $\mathbb{C}_{\text{names}}$ **Common components in both analyses**: these are common components found in both analyses. Note that $\mathbb{C}_{\text{names}}$ only hold component names, as there are actually two sets $\mathbb{C}_A$ and $\mathbb{C}_B$ holding potentially (but not necessarily) different models $m_A(c) \in \mathbb{C}_A$ and $m_B(c) \in \mathbb{C}_B$ describing the same components c.
– Set $\mathbb{D}$ **Common components, all interfaces from both models**: these are the same components as described by $\mathbb{C}$, but with a redefinition of their list of interfaces as the sum of the interfaces from both analyses: $\mathbb{D} = \{m_A(c) \oplus m_B(c) \mid c \in \mathbb{C}\}$. Note that this set is already the result of a first merging step common to both strategies.

The two merging strategies considered are: **Strategy A**: Inclusive merge. This computes the union set of the three sets $\mathbb{A}$ $\mathbb{B}$ and $\mathbb{D}$ above. This is the less restrictive one and it is optimal for an exhaustive debug of systems, but it is the less recommended strategy to diagnose systems at runtime, since it will likely contain duplicated components and interfaces for those cases where the static analyzer was not able to extract correctly the names of the interfaces. **Strategy B**: Enriching merge. With this strategy, we will complement the result of the static analysis by adding to its components the missed interfaces, which will increase considerably the amount of data information without corrupting the models with fictitious (filesystem) information. Formally this computes the union of $\mathbb{A}$ and $\mathbb{D}$. That means that the resulted model supports all our tools and features (e.g., it can be used to deploy the entire system). We can imagine that this strategy can be used for alternative purposes; for example, without a previous static analysis, the user can just define a system as a list of components (only the names) and let this feature to fill out the interfaces of each component, which is a great and easy method to define new robot specifications.

In Sect. 8, the reader can find a detailed analysis of the metrics of the data from all the strategies obtained by a real experiment with a large system.

7 Runtime target: model adaptation and leveraging models at runtime

In this section, we analyze the usability and validity of our metamodels to reach the new target goals for this paper: allow the detection of errors not only at design but also at runtime.

First of all, we have to evaluate the usability of our models to describe the peculiarities of a ROS at runtime. In other words, we aim to answer the following research question:

RQ2 *Can a component-based MDE approach with the same metamodel cover all the peculiarities that characterize a Robot Operating System application at design* **and** *at runtime? What relevant information has to be considered to analyze also a system during its execution?*
If we apply the previous research question to our research and our previous efforts, we have to reconsider here if our metamodels are complete enough to cover the necessary information or if we missed relevant runtime particularities.

Based on this evaluation, our next action was the adaptation of the metamodels to solve the found shortcomings. Finally, we present our first features, that thanks to the rest of our approach and the new adaptation and combination of models, leverage our effort to monitor and diagnose ROS systems at runtime.

7.1 Suitability for runtime analysis

We have demonstrated in previous publications [33,34] the validity of our metamodels to design ROS robot applications, and we have consolidated our concept with all the complementary tools (see Sect. 3.2) that shows to the ROS community the benefits of the use of MDE technologies. Although the main target of all our efforts until now was design time, now that we covered this first phase of the software development, we recently started to explore the potentials of our research during the execution of the system.

Our first novel contribution is a tool to diagnose a running application using our models , which helps us also to find the deficiencies of the metamodels by using static analysis code results to check ROS system at runtime. Unsurprisingly, this pointed out a deficiency of our models to deal with one of the most conflictive tools of ROS, the parameters and the parameter server [11]. This shared and multivariable dictionary can be used to exchange data between all the programs with access to the ROS master. These data can be exchanged dynamically and at runtime, and there is no need to define them at design time. Even more a parameter in ROS can be defined without the specification of its type attribute (i.e., Boolean, String…), which gives freedom to the user to set any value during the execution of the program. Depending on the software architecture design, these parameters can influence the behavior of the system considerably and so far there is not a optimal approach to validate them.

The new ROS 2 version has set more restrictions to the developers concerning the definition of parameters, which, from our point of view, is an appropriate action and a step forward on code quality and systems validation.

7.2 Metamodels adaptation to support runtime systems

The migration of our full tooling to ROS 2 is de facto the next step of our work. ROS 2 is already consolidated in terms of available software packages and the community of users supporting it. The extension of the metamodels to support ROS 2 will be minimal. In fact, we have already successfully integrated some code generators. Then, and because in ROS (or ROS 1) we miss completely a proper definition of parameters, we have adapted our metamodels to be able to support the description of the parameters and validate their values at runtime in the same way that ROS 2 expects to use them. And lastly, we updated the monitor extractor at runtime to support this new model extension and be able to automatically extract all the shared information of the parameter server dictionary.

This new extension allows the user to describe also parameters at design time and to set to them a type and a default value. We also give to the user the option to re-define the value of the parameter at the application specific level, like ROS does at runtime or design time using its deployment artifacts. In parallel, we added to our DSLs validators the appropriate rules to check the correctness of the redefinition of parameters. In the next section, we introduce the first leverage tools that we created to validate this new extension of our metamodels.

7.3 Leveraging models at runtime

After the study of the current state of the art to diagnose ROS systems during their execution, we found the use of MDE as a good fit for this domain, since it can simplify and significantly optimize error debugging and even facilitate the integration of error-handling techniques.

The final goal of our efforts is to develop a diagnosis and monitoring framework whose main objective is the integration of existing open-source ROS based tools as observers. This framework is directly linked to the ROS models, using them to (1) evaluate continually if the components and interfaces running on the system are the expected ones (we called this tool **ROS Graph observer**) and (2) to select from the models the sensitive properties to be monitored during the design of the application and auto-generate the deployable code for its inspection (we called this feature **properties observer**). Although the property observers do not address the functional-safety-concerns regarding robotics software [17], these property observers, which are independent of the application program written in Python, can be used to diagnose system-level properties by monitoring component-level properties.

The basis of the **ROS Graph observer** is our runtime introspection parser and extractor, which can return at any time the list of the current nodes running on our systems, the interfaces that connect them and all the parameters set together with their values. All this information is obtained as formally structured through our metamodels. We built a generic system diagnostics tool (the rosgraph_monitor) upon it that does not require an application-specific configuration.

Figure 4 shows a simplified overview of the approach. First, we give to the rosgraph_monitor a *desired.rossystem* model specification file of our desired system, which contains the list of the nodes, interfaces, and /or parameters that we expect to be running on our robot, and second we call periodically the *ros_graph_parser* to output the current status *current.rossystem* of the system. Finally, we interpret and compare for both models (1) the nodes (components), (2) the interfaces contained within them (component ports) and the established connections and (3) the type of the parameters and their fixed values. Lastly, it outputs the result of this analysis.

For a full integration of ROS and our Xtext languages, we implemented a pyparsing-based interpreter that makes all the information contained on our models accessible from ROS native code. Also, the result of the analysis is published as a common ROS topic that can be interpreted and visualized with the common ROS diagnostics tools or sent to an error-handling tool to deal with the fail and reconfigure the system.

The de facto use of this feature is to verify if a set of essential components have been started and are running. Printing an error in case one of them died or does not present the expected interfaces. This approach simplifies significantly the diagnosis of a system at runtime, without the need of the development of a new specific diagnostic program for each component or having this part embedded on the driver. Moreover, we discovered other uses of this feature; currently, we use it to quickly and easily detect if a robot setup has all the requirements (in terms of interfaces) to perform a concrete task and if it is compatible with other existing software libraries. For these cases, the desired RosSystem file, that we use as input, describes through our models format the specifications for a concrete software application and the ROS Graph observer will evaluate if the current running system

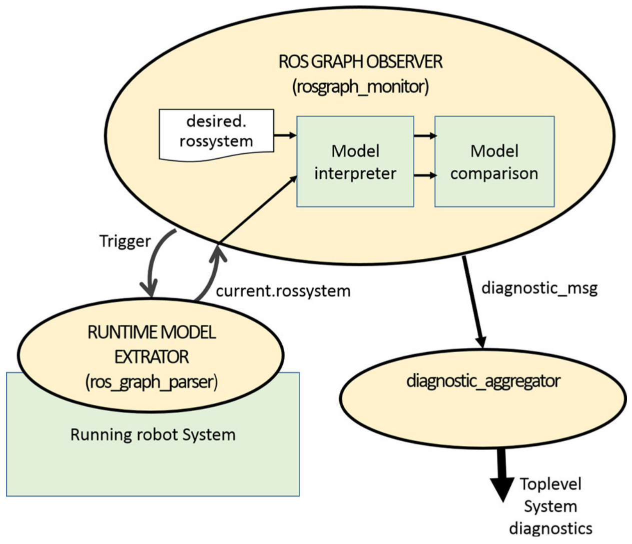


Fig. 4 Architecture overview of the ROS graph observer approach. The flow of the information begins with the runtime running system and its model equivalent obtained thanks to the *ros_graph_parser*, comparing this data with the *desired.rossystem*; this solution publishes a *diagnostic_msg* with the status of the current execution

fulfills the requirements. If it does not, a detailed diagnostics message will list the missing interfaces. This can also be extended to proof parameters and parameters values for a more precise study of compatibility.

The previously described tool checks whether the components, their ports, their connections, and the parameters that we expect during the design of our application are present during its execution, but we can move one step forward. At runtime, the values of the objects of the exchanged information are also available, we just have to subscribe to the running interfaces, and our models contain a formal description of the type of this information. Upon this, we recently developed a new plug-in for our Tooling that allows the user to set at design time the thresholds for a number or the concrete expected value of the information objects. This plug-in will auto-generate, as outcome, a ROS python node that will publish a diagnose error if the value of one object property infringes the user-defined range.

For example, thanks to our metamodels, we know that an arm driver expects as command an array of *Floats* that represent the desired position for each joint. With this new plug-in, the user can select the arm command interface and set the maximum and minimum values for the position of each joint. The auto-generated observer, executed together with the rest of the application, will publish an error message if one joint position command exceed the limits. Even more, the generated code is implemented in such way that it can be easily extended to, not only publish the diagnostics messages, but also call other functions, like in this case could be stop or reject the move command action.

This novel contribution aims to be a first step to bridge our MDE efforts with error-handling and self-adaptation technologies. We propose here an approach where using a single metamodel we can describe the desired specifications of the system and generate its full deployment artifacts together with a set of properties observers. The observers' outputs could feed an error-handling software, that, analyzing the data, can identify and react to behavior anomalies.

8 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 project SeRoNet.

In order to practically demonstrate the concrete technical contributions of this paper, we report in this section a use case based on the commercially deployed service robot Care-O-bot4 or cob4 (Fig. 5). Given its complexity, we can consider it an adequately representative example for service robotics applications.

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, in order to facilitate the understanding of the next subsections and the vocabulary used to describe the experiments, we firstly list the set of publicly available repositories from which to source the companion material:

Fig. 5 Care-O-bot4 full robot. Developed by Mojin Robotics

- *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.[7]
- *ROS Cloud tool* This repository contains the backend code of the web interface publicly available to extract models (i.e., ros-model.seronet-project.de). 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.[8]
- *ROS Graph parser* This repository holds the ROS package used for the monitoring and extraction of models at runtime.[9]
- *ROS Graph monitor* This repository holds the ROS node that parse and compare a desired specification model of a system and the actual runtime status from the ros_graph_parser. And then publish a diagnostic message as result of the comparison.[10]
- *ROS Experiments* This repository contains the results of different analysis, including the full results of the experiment explained in this section corresponding to the version cob4-25.[11]

To perform the system experiments, we used the software description of the drivers that manage the hardware of the cob4-25. 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; three 3D cameras for visualization tasks; components like light, mimic control, and sound for general robot–user interaction. Figure 5 shows the real aspect of one of the robot of the series.

This section is divided into four parts: **first** we focus the experiment on the static code analysis to auto-extract models and show an example of its benefits together with the web cloud infrastructure to analyze and find a component replacement for our robot, **second** we show how our auto-extraction methods perform the analysis for a complete large ROS system and compare the obtained results; then, **third**, we explore the use, with the two strategies, of our new feature to combine both methods. Finally, **fourth**, we introduce

[7] https://github.com/ipa320/ros-model.

[8] https://github.com/ipa320/ros-model-cloud.

[9] https://github.com/ipa-led/ros_graph_parser.

[10] https://github.com/ipa-hsd/rosgraph_monitor.

[11] https://github.com/ipa-nhg/ros-model-experiments.

our first experiments using models to diagnose a ROS large system at runtime.

8.1 Components model extraction through static code analysis

8.1.1 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, in this case, 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.

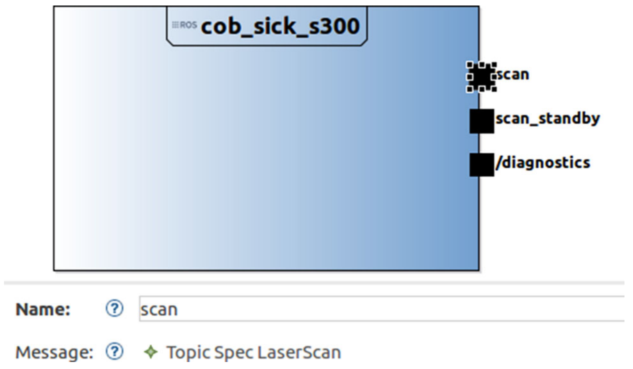
For this concrete walk-through tutorial, we opted for the cloud solution, giving as input (the information of one of the scanners node):

- **Git repository** 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 shown in Fig. 6

```
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



Name: scan

Message: Topic Spec LaserScan

Fig. 6 Tooling visualization of the auto-generated model for a Sick s300 scanner driver

8.1.2 Leveraging static code analysis to extract common specifications: component replacement use

To show the potential of large-scale code analysis made possible by our cloud system, we choose a typical use case in robotics: a system integrator in need of replacing one of the components of a system. The process of replacing a robot component starts with an evaluation of the alternatives in the market, firstly, in terms of hardware (physical dimensions and electronic requirements), and secondly, 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 [1]. From this list, we filtered 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 have obtained the model for each of them using our cloud tooling for the extraction (i.e., http://ros-model.seronet-project.de) with the following input list:

HLS-LFCD LDS:

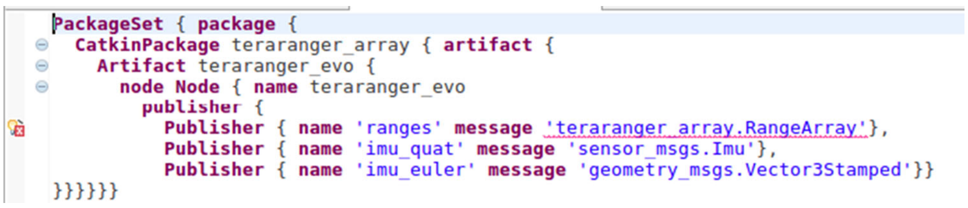
- *Git* github.com/robotis-git/hls_lfcd_lds_driver
- *Package* hls_lfcd_lds_driver
- *Node name* hlds_laser_publisher

Hokuyo:

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

Pepperl Fuchs r2000:

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



```
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. 7 Teraranger Evo auto-generated model imported on the tooling which shows an error because of the use of a non-common ROS message pattern for a publisher interface

Rplidar:

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

Sick Safety Scanners:

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

Teraranger Evo:

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

Neato XV-11:

- *Git* 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.

The **first** check is the evaluation of the use of standard messages. This check was not passed by the *Teraranger Evo* (as shown in Fig. 7) 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 , 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 2 Result, generated by the tooling, of the comparison of the hlds_laser_publisher with the common specification pattern for a 2D Laser scanner

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 (*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 (Sect. 5.2). All the analyzed package except the *Teraranger Evo* passed this check; Lst. 2 shows the output we obtained for all the drivers that fulfilled the requirement of publishing their output through the standard message type *sensor_msgs/LaserScan*. Lst. 3 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 3 Result, generated by the tooling, of the comparison of the teraranger_evo with the common specification pattern for a 2D Laser scanner

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. 4. 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 4 result of the comparison of the hokuyo_node with the sick_s300 model generated by the tooling

Thanks to the static code analysis, the open-source nature of ROS and the tools we developed upon our metamodels to check and compare models, we demonstrated how the use of a MDE approach for ROS common software simplifies significantly the identification of common design patterns.

8.2 Automatic extraction of ROS large system models

8.2.1 System model extraction through static code analysis

The previous 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, written in XML format, are used in ROS to start together several programs.

Figure 8 shows an overview of the approach running on our cloud system within a Docker container. 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 second obtains for each single package the GitHub URL that holds the source code (command *roslocate info*). Once we have this information, we checked out all the source code to our workspace and compile all these packages each package in an isolated environment , option recommended as the cloned packages can likely have different make system variants (catkin and non-catkin). 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, 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, together with a successful build of all the packages, we invoke the source code extractor (same code than in the previous step of the experiment) to generate automatically:

– a set of component models (**.ros** files), including the model of the Sick S300, i.e., Lst. 1) *ros-model-experiments/cob4-25/cob4-25_static/rosnodes*
– a RosSystem model 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. 5 *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 5 Sample of the RosSystem model generated by the static code analyzer

Examining 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..), getting the value of these arguments the controller manager set at runtime for all its commands the name and the type of the interfaces. This case is impossible to be supported by a static code analyzer. To momentarily solve this issue, also knowing that once the type of controller is defined the delivered interfaces are fix, we defined a special plug-in for our extractor that checks the arguments of the *con-*

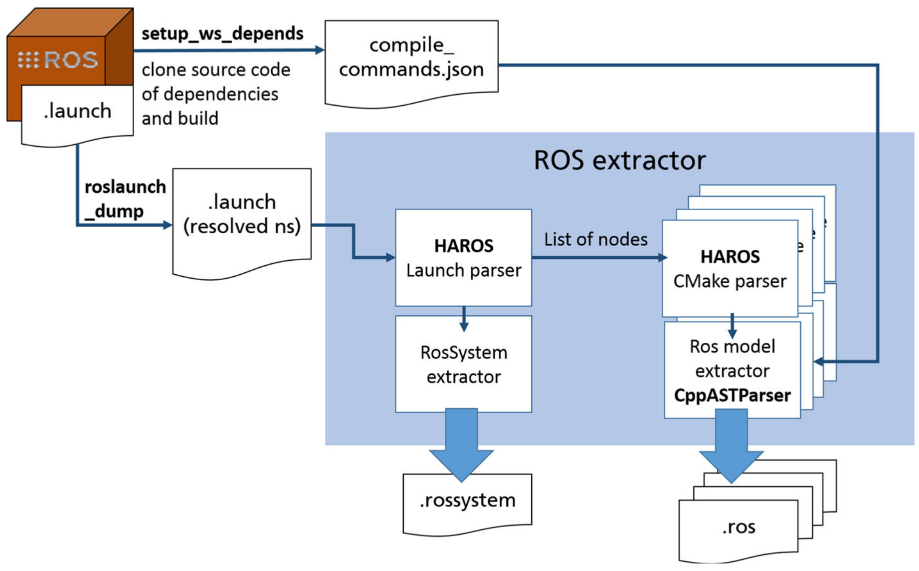


Fig. 8 Diagram of the approach of system model extraction through static code analysis

troller_manager node using a predefined 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.

8.2.2 System model extraction through runtime monitor

For this part of the experiment, we used the hardware of the real robot (Fig. 5) and started all the components used for the previous section analysis, in order to obtain comparable results.

Starting the same launch file and initializing all actuators (i.e., base, torso, sensorring and both arms) to make their commands available, we called our ROS graph parser (Sect. 4.2) to extract the model of the system. The result of this experiment is two files:

– the fictitious component model. A .ros file model 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 *cob4-25/cob4-25_runtime/rosnodes/dump.ros*.
– a RosSystem model file 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. 6 *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 6 Sample of the RosSystem model generated by the runtime monitor

A peculiarity of the result of this analysis is the addition of a fictitious node, called *parameters_node*, which includes a complete list of all the parameters set on the system during the execution including the value of all of them. This new feature is very useful for the debug of the system; in the case of this Care-O-bot experiment, we found a total of 1159 parameters.

8.2.3 Comparison of system extraction through static code analysis and runtime monitoring

In Sect. 4, we listed the technical limitations of each approach. Before we analyze the obtained numbers we have to add, in favor of HAROS and the static analysis, that Care-O-bot, the case of study, is a very complex system which modularity enforces the definition of many of the interfaces through arguments. The support of this configuration by a

Table 1 Comparison of the number of components and the different types of interfaces found by the both extraction methods

	Static analysis	Runtime analysis
Components	69	75
Topic Publisher	146	259
Topic Subscriber	36	105
Service Server	57	153
Service Client	20	0
Action Client	6	7
Action Server	6	11
Total interfaces	271	535

static code analysis tool is limited and hard to be totally integrated.

Table 1 shows the quantitative comparison of both approaches in terms of number of components and interfaces found.

In total with the runtime approach, we found the 76 nodes listed on the launch file, expected result as ROS starts all the nodes included on a launch file automatically once it is called and we performed the experiment with a fully operative robot. However, we cannot expect that rate for the interfaces found, for example, the case of the Service Clients, and all the interfaces activated within callbacks, that if not called (likely the case) during the monitoring time, were fully ignored.

If we analyze the results for the static code analysis, we obtained 69 components (a total rate of a 92%). This means that for Care-O-bot we could find for the 92% of the components an open-source code version using just the ROS distro mechanisms. Even more important, for all of them the code could successfully build in a clean image of the operating system, where all the packages dependencies where found and installed automatically and all the quality requirements needed to analyze statically the code passed successfully. From this data, we can extract two main conclusions, (1) the design leveraging HAROS, the tools and the docker container solution we propose is appropriate and valid to be used to analyze large systems and (2) the full Care-O-bot software stacks (that conglomerates more than 100 ROS packages) in spite of, or maybe thanks to, its open-source nature evidence a high grade of code quality, remember that at the end we analyze C++ and Python code and how it is designed and written.

If we use the obtained data to answer our **RQ1**: "How complete is the information we can automatically obtain (from code-to-model extraction methods) to describe a large Robot Operating System application as a set of modular and reusable components and the interaction interfaces among them? Is there incorrect information?" we see that, firstly, in terms of informativeness the models extracted through the static code analysis (e.g., Lst. 1) are truthful and the ROS metamodels structures fully fulfilled, remember that we considered this as a premise requirement of our research question. 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 their interfaces are already remapped. The assignment of a namespace is done during deployment phase (i.e., for the family of metamodels, included in the RosSystem model file, see Lst. 5).

Secondly if we consider the usability and re-usability, these listed issues, mainly the lack of the filesystem information, make 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.

This conclusion, the lack of relevant data from the runtime introspection set the new direction of our research exploring the possibilities to combine the results of the extraction methods and, therefore, the benefits of both.

Unsurprisingly, the ROS Graph monitor is able to find more interfaces than the static analysis, by comparing the number of detected interfaces through the runtime monitor (535) and through the static code analyzer (271). A deep analysis of the data and the cases where the static code introspection failed answered our **RQ2**: "Can a component-based MDE approach with the same metamodel cover all the peculiarities that characterize a Robot Operating System application at design and at runtime? What relevant information has to be considered to analyze also a system during its execution?", and confirmed our suspicions; in ROS for large, and likely modular, systems the parameters and their set and unset at runtime play a very important role and change completely the behavior and the interactions between the nodes once the system is deployed. Giving some numbers, for eight components (from a total of 35 software packages) all their interfaces are defined by the set of parameters and the obtained models extracted analyzing their source code are empty. Also, for a total of 18 interfaces we couldn't determine their names, which are passed as a string parameter. This is a very common practice in ROS for generic components, as the use of parameters facilitates the re-usability and easy adaptation of a node to different applications.

After this study, we decided to update our models to support properly the definition of parameters and improve all our runtime introspection tools for a better analysis of the ROS parameters particularities.

8.3 Combination of the results of both extraction methods

This step of the experiment shows how our combination feature can be used to compose the data obtained by both extraction methods and obtain a more precise model of a system. Figure 9 represents the approach we used for this demonstration, where we took as inputs the same models we compared in the previous step, extracted automatically by our both extraction methods and that targeted the same system. The result will be an unique RosSystem file which data will depend on the criteria selected by the user.

For this experiment, we used the two strategies we explained in detail in Sect. 6.2, i.e.,

- *Strategy A or inclusive merge* which results in the union of the components of the static code analysis with the ones found by the runtime monitoring tool, by the common components this strategy merges all those found.
- *Strategy B or enriching merge* where all the components from the static analysis are present and only for those also found by the runtime analysis we add the missed interfaces.

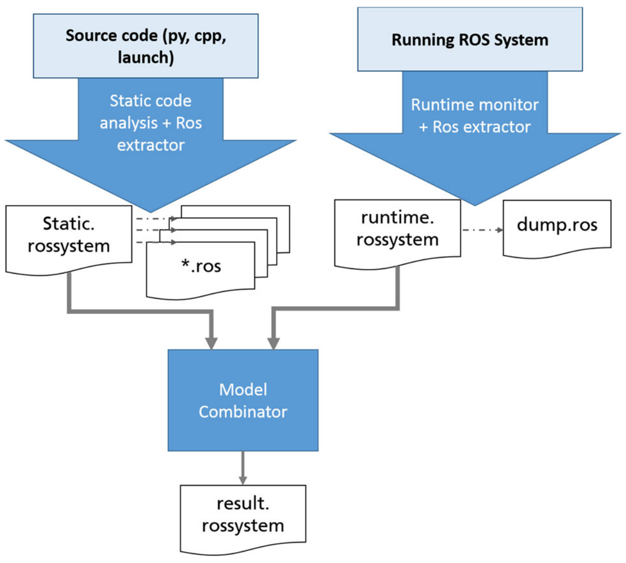


Fig. 9 Architecture overview of the feature to combine two models for Sect. 8.3 case, i.e., combine models auto-generated using our two extraction methods for the same system, the cob4-25 robot

Table 2 Comparison of the number of components and interfaces obtained by the both model combination criteria applied to the results from a static code analysis and the runtime monitor for the cob4-25 robot system. Complemented, marked in bold, with the data obtained for the model obtained from the combination

	Strategy A	Strategy B
Components static analysis (from previous section)	69	69
Components runtime analysis (from previous section)	75	75
Components of the resulted model	100	69
Duplicated components of the resulted model	25	0
Interfaces of the resulted model	775	676

The entire results of this experiment are publicly available under *ros-model-experiments/cob4-25/*. In Table 2, we summarize them quantitatively.

If we analyze the data, we see that with the *strategy A* we get many duplicated components interfaces, whose names (likely because they were re-assigned using a parameter from the parameter server or the value dynamically passed through code functions) do not correspond to the final assignment at runtime. If the purpose of the user is to make an exhaustive debug of the system, this result is the most complete that can be obtained. Filtering the data of the resulting model, all the errors introduced either during the design or at the execution of the application can be detected.

With the *strategy B*, on the other hand, we successfully completed a large portion of the interfaces missed by the static analysis (676 in total, while by the static code analysis we only detected 271), and without having duplicated components. Even better, for all the components of the final resulting model we have complete information, i.e., all the fields of our metamodels are filled including the real filesystem information.

If we link the experiment results with our **RQ1**: "How complete is the information we can automatically obtain (from code-to-model extraction methods) to describe a large Robot Operating System application as a set of modular and reusable components and the interaction interfaces among them? Is there incorrect information?", we see that with the strategy B (the enriching merge) we obtained a final RosSystem model that contains 69 of the 75 components started on the real robot and all their interfaces (that correspond to a 92% of the total) and for all of them we have all the required information, the related to the computational graph but also the filesystem part. This makes the resulting model compatible with all our ROS tooling features and re-usable for other applications and even valid for its composition with other systems or frameworks.

8.4 Leveraging models at runtime

8.4.1 Auto-diagnosis of a system evaluating its computational graph at runtime

To show the use of our monitoring tool, we described the list of the most common hardware drivers of the Care-O-bot4 as a RosSystem model (file available under cob4-25_desired_components.rossystem) and used it as the specification of the desired system for our experiment. When all the necessary hardware is connected and is in running state, the diagnostics publisher will contain the OK status message like shown in Lst. 7.

```
status:
    level: 0
    name: "ROS Graph"
    message: "running OK"
    hardware_id: ''
    values: []
```

Listing 7 Diagnostics message for the rosgraph_monitor

If one of the desired nodes is not running, it is possible that the system may not function as intended. These nodes are marked as *Missing nodes* with an operation level 2 (ERROR). Evaluating the common specification desired file for the Care-O-bot4 family on the Care-O-bot4-25 version we detected, firstly, that this robot distro does not have a microphone driver, and therefore, the sound component is not available. Another test we made to prove this feature was to physically disconnect one of the robot's cameras, which does not allow to start the camera driver at startup. As expected, the monitoring tool marks this also as a missing node like can be seen in Lst. 8.

```
status:
   level: 2
   name: "sound"
   message: "Missing node"
   hardware_id: ''
   values: []
status:
   level: 2
   name: "/torso_cam3d_right/realsense2_camera"
   message: "Missing node"
   hardware_id: ''
   values: []
```

Listing 8 Diagnostics message for the rosgraph_monitor if a node is missing, sound example

Although we strongly recommend the implementation of diagnostics messages as output for every ROS driver, with this experiment we demonstrate that, without the extra development effort or a custom configuration, we are also able to detect component failures. But more importantly, the head camera scenario shows an use case so far unsupported; the camera driver was never started, because it has a precondition to detect the physical hardware first, and therefore the diagnostics message from the software driver was never published and the error never showed, but still our solution could detect it.

Additionally, when components not mentioned as desired are detected in the graph, it is not an error, but if the user needs to check the extra components started on the robot, the monitoring framework marks for these cases the components as *Additional nodes* with an operation level 1 (WARN). These operation level can be used in ROS to easily filter or sort the diagnostics message by their severity.

8.4.2 Auto-evaluation of the running system and its capabilities

For the experiments, we decided to use the common low-level functionalities of Care-O-bot; however, on ROS there are open-source libraries to perform high-level capabilities that can be applicable to any robot setup to perform complex tasks. The most common robotic capabilities are navigation, perception and manipulation and scenarios based on the coordination of them. One of the biggest advantages of ROS is the modularity, adaptability and re-usability of these software components, but to run these capabilities, your robot has to offer a concrete set of interface that the high-level functionality needs to get or command information. Currently the identification of the interfaces and the check of their presence has to be done manually. With our approach, we found a way to semi-automate the process, being able to analyze the running system and automatically detect whether it is suitable to run a concrete application. Going deep into the technical implementation, we created RosSystem models to describe formally the specifications of a capability, for example, to navigate a robot the ROS navigation stack requires the information from a laser scanner, the kinematics tree of the robot and its odometry. Once it computes of all of this data, it will publish a velocity twist command to the base. Lst. 9 shows the formal model for this description for the Care-O-bot4 (cob4) target.

```
RosSystem { Name 'cob4-25_navigation_capability'
 RosComponents (
 ComponentInterface { name 'navigation'
  RosPublishers {
   RosPublisher '/scan_unified' {RefPublisher 'cob4
     /../scan_unified'},
  RosPublisher '/tf' {RefPublisher 'cob4/../tf'},
  RosPublisher '/base/odometry_controller/odometry'
    {RefPublisher 'cob4/../odometry'}}
  RosSubscribers {
  RosSubscriber '/base/twist_controller/command' {
     RefSubscriber 'cob4/../command'}
 }}
)}
```

Listing 9 RosSystem model excerpt corresponding to the ROS navigation capability specification

Using the previous model as the desired system for our ROS Graph observer feature, we get automatically a message with the result of the analysis. If it fails we also point to the missing interface or interfaces, Lst. 10 present and example of the diagnostic output.

```
status:
  level: 2
  name: "cob4-25_navigation_capability"
  message: "Capability incompatible"
  hardware_id: ''
  values:
    -
    key: "publishers"
    value: "tf"
```

Listing 10 rosgraph_monitor diagnostic output message when an interface to run the Navigation capability is missing in the system

9 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 and the identification of de facto standard practices. The reuse of such practices 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 to the model-based verification of manually written ROS code and push toward 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, for example, 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 continue our growing effort on monitoring and diagnostics at runtime, by adding further system properties observers and integrating reasoning models to be able to automatically handle errors during the execution of an application.

Acknowledgements 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. Other parts of this work have been supported by the European Commission's Horizon 2020 Programme under the Grant Agreements No. 732287 (ROSIN). And 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).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Catalog of ROS supported sensors. http://wiki.ros.org/Sensors. Accessed 29 Mar 2021
2. COB monitoring source code-GitHub link. https://github.com/ipa320/cob_command_tools/tree/indigo_dev/cob_monitoring. Accessed 29 Mar 2021
3. DiagnosticStatus message definition. http://docs.ros.org/api/diagnostic_msgs/html/msg/DiagnosticStatus.html. Accessed 29 Mar 2021

4. Docker project website. https://www.docker.com/. Accessed 25 Apr 2019
5. IEEE RAS Technical Committee for Software Engineering for Robotics and Automation. https://www.ieee-ras.org/software-engineering-for-robotics-andautomation. Accessed 29 Mar 2021
6. Industrie 4.0 an der Börse: Software lukrativer als Hardware. https://www.wiwo.de/finanzen/geldanlage/industrie-4-0-an-der-boerse-softwarelukrativer-als-hardware/14452080-3.html. Accessed 29 Mar 2021
7. RobMoSys—Composable Models and Software. https://robmosys.eu/. Accessed 29 Mar 2021
8. Robotics Software Engineer Nanodegree Program. https://eu.udacity.com/course/robotics-software-engineer--nd209. Accessed 29 Mar 2021
9. ROS metrics wiki site. http://wiki.ros.org/Metrics. Accessed 29 Mar 2021
10. ROS: Robot Operating System. http://www.ros.org/. Accessed 25 Apr 2019
11. ROS wiki link to the parameter server. http://wiki.ros.org/Parameter20Server. Accessed 29 Mar 2021
12. ROS2: Developers guide. https://index.ros.org/doc/ros2/Contributing/Developer-Guide/. Accessed 28 Feb 2020
13. ScalABLE 4.0 project website. https://www.scalable40.eu. Accessed 29 Mar 2021
14. SeRoNet project website. https://www.seronet-projekt.de. Accessed 29 Mar 2021
15. Simplified interface for syntax trees and program models. https://github.com/gitafsantos/bonsai. Accessed 29 Mar 2021
16. Unified Architecture—OPC Foundation. https://opcfoundation.org/about/opctechnologies/opc-ua. Accessed 29 Mar 2021
17. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Towards rule-based dynamic safety monitoring for mobile robots. In: Simulation, Modeling, and Programming for Autonomous Robots, pp. 207–218. Springer International Publishing, Cham (2014)
18. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor Franco, J.A., Álvarez Torres, B.: V3cmm: a 3-view component meta-model for model-driven robotic software development. J. Softw. Eng. Robot. **1**, 3–17 (2010)
19. Ando, N., Suehiro, T., Kotoku, T.: A software platform for component based RT-system development: OpenRTM-Aist. In: Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 87–98. Springer, Berlin (2008)
20. Bihlmaier, A., Wörn, H.: Increasing ROS reliability and safety through advanced introspection capabilities. Informatik **2014**, 1319–1326 (2014)
21. Brugali, D., Hochgeschwender, N.: Software product line engineering for robotic perception systems. Int. J. Semant. Comput. **12**, 89–107 (2018). https://doi.org/10.1142/S1793351X18400056
22. Brugali, D., Scandurra, P.: Component-based robotic engineering (part I) [tutorial]. IEEE Robot. Autom. Mag. **16**(4), 84–96 (2009)
23. Brugali, D., Shakhimardanov, A.: Component-based robotic engineering (Part II). IEEE Robot. Autom. Mag. **17**(1), 100–112 (2010). https://doi.org/10.1109/MRA.2010.935798
24. Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., Brugali, D.: The BRICS component model: a model-based development paradigm for complex robotics software systems. In: ACM Symposium on Applied Computing (SAC), SAC '13, pp. 1758–1764. ACM, New York (2013). https://doi.org/10.1145/2480362.2480693
25. Bubeck, A., Weisshardt, F., Verl, A.: BRIDE—a toolchain for framework-independent development of industrial service robot applications. In: International Symposium on Robotics (ISR/Robotik), pp. 1–6 (2014)
26. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 149–160. Springer, Berlin (2012)
27. Estevez, E., Sánchez, A., García, J., Ortega, J.: ART2ool: a model-driven framework to generate target code for robot handling tasks. Int. J. Adv. Manuf. Technol. **97**, 1–13 (2018). https://doi.org/10.1007/s00170-018-1976-z
28. Estevez, E., Sánchez-García, A., Gámez-García, J., Gómez-Ortega, J., Satorres, S.: A novel model-driven approach to support development cycle of robotic systems. Int. J. Adv. Manuf. Technol. **82**, 737–751 (2016).https://doi.org/10.1007/s00170-015-7396-4
29. Foote, T.: ROS Community Metrics Report. http://download.ros.org/downloads/metrics/metrics-report-2019-07.pdf (2019)
30. Gerkey, B., Stoy, K., Vaughan, R.T.: Player robot server. Tech. rep., Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California (November 2000)
31. Gherardi, L., Hunziker, D., Mohanarajah, G.: A software product line approach for configuring cloud robotics applications. In: IEEE International Conference on Cloud Computing (CLOUD), pp. 745–752. https://doi.org/10.1109/CLOUD.2014.104 (2014)
32. Hägele, M.: Robots conquer the world [turning point]. IEEE Robot. Autom. Mag. **23**(1), 118–120 (2016). https://doi.org/10.1109/MRA.2015.2512741
33. Hammoudeh Garcia, N., Deval, L., Lüdtke, M., Santos, A., Kahl, B., Bordignon, M.: Bootstrapping mde development from ros manual code - part 2: Model generation. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 95–105. https://doi.org/10.1109/MODELS.2019.00-11 (2019)
34. Hammoudeh Garcia, N., Lüdtke, M., Kortik, S., Kahl, B., Bordignon, M.: Bootstrapping mde development from ROS manual code—part 1: metamodeling. In: IEEE International Conference on Robotic Computing (IRC), pp. 329–336. https://doi.org/10.1109/IRC.2019.00060 (2019)
35. Hammoudeh Garcia, N., Lüdtke, M., Kortik, S., Kahl, B., Bordignon, M.: Bootstrapping MDE development from ROS manual code. Int. J. Robot. Comput. (IJRC) **2** (2020)
36. Kilgo, P., Syriani, E., Anderson, M.: A visual modeling language for RDIS and ROS nodes using atom3. In: Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 125–136. Springer, Berlin (2012)
37. Kittmann, R., Fröhlich, T., Schäfer, J., Reiser, U., Weißhardt, F., Haug, A.: Let me Introduce Myself: I am Care-O-bot 4, a Gentleman Robot. In: Mensch und Computer, pp. 223–232. De Gruyter Oldenbourg, Berlin (2015)
38. OMG: Deployment and Configuration of Component-Based Distributed Applications Specification Version 4.0. Object Management Group (2006)
39. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: IEEE International Conference on Robotics and Automation (ICRA)—Workshop on Open Source Software (2009)
40. Santos, A., Cunha, A., Macedo, N.: Static-time extraction and analysis of the ROS computation graph. In: IEEE International Conference on Robotic Computing (IRC), pp. 62–69 (2019). https://doi.org/10.1109/IRC.2019.00018
41. Santos, A., Cunha, A., Macedo, N., Arrais, R., dos Santos, F.N.: Mining the usage patterns of ROS primitives. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3855–3860 (2017). https://doi.org/10.1109/IROS.2017.8206237
42. Santos, A., Cunha, A., Macedo, N., Lourenço, C.: A framework for quality assessment of ROS repositories. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4491–4496. IEEE. https://doi.org/10.1109/IROS.2016.7759661 (2016)
43. Schlegel, C., Worz, R.: The software framework SMARTSOFT for implementing sensorimotor systems. In: IEEE/RSJ International

Conference on Intelligent Robots and Systems (IROS), vol. 3, pp. 1610–1616. https://doi.org/10.1109/IROS.1999.811709 (1999)
44. Soetens, P.: A software framework for real-time and distributed robot and machine control. Ph.D. thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf (2006)
45. Stampfer, D., Lotz, A., Lutz, M., Schlegel, C.: The smartmdsd toolchain: an integrated mdsd workflow and integrated development environment (IDE) for robotics software. J. Softw. Eng. Robot. (JOSER) **7**, 3–19 (2016)
46. Zaman, S., Steinbauer, G., Maurer, J., Lepej, P., Uran, S.: An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 482–489. https://doi.org/10.1109/ICRA.2013.6630618 (2013)
47. Zhang, L., Merrifield, R., Deguet, A., Yang, G.Z.: Powering the world's robots—10 years of ROS. Sci. Robot. (2017). https://doi.org/10.1126/scirobotics.aar1868

Nadia Hammoudeh García obtained in 2011 her diploma degree in Telecommunication Engineering from the Universidad Politecnica de Madrid. In December 2011, she joined as research fellow the Department for Robots and Assistive Systems of Fraunhofer IPA in Stuttgart, Germany. The first 5 years at IPA, among other publicly funded and private R&D projects, she was involved on the development of the service robot Care-O-bot4, that nowadays is a commercial product and which is fully developed with open-source software, the Robot Operating System (ROS). Her core activities within the team were related to software modularity, composability and reconfigurability. Since 2017, she works as project manager. She focusses her research on the adaptation of open-source robotics software to be transferred to industrial environments. More specifically in the fields of software quality assurance and the simplification of the software design and development process by the adoption of model-driven engineering techniques.

Harshavardhan Deshpande M.Sc. received his masters degree in Mechatronics from Fachhochschule Aachen in 2017. In March 2018, he joined the Department of Robots and Assistive Systems at Fraunhofer IPA, Stuttgart, Germany, as a research engineer. He has since been involved with contributing to the Robot Operating System (ROS), an open-source project. These projects, along with other publicly- and privately-funded projects he worked on, had a strong focus on software quality assurance. These experiences led him to his current research area in runtime and diagnosis of open-source robotics software used specifically in industrial environments.

André Santos has a master degree in Software Engineering from the University of Minho. He is doing his Ph.D., at the same University, on how to bring standard Software Engineering practices to the robotics ecosystem, specifically that of the Robot Operating System, in order to verify critical architectural and behavioural properties.

Dr. Björn Kahl completed his studies on theoretical physics at Kiel University Christian-Albrechts-Universität zu Kiel, Germany, in 2000 with the diploma degree. He then moved to the field of robotics and received his doctoral degree in 2007 from University of Bayreuth, Germany, working on fast simulation of contact forces in assembly tasks. He held multiple research and teaching positions at Bonn-Rhein-Sieg University of Applied Sciences in the Autonomous Systems program and supervised development of complex software systems for robotics and AI in international projects. Since 2012, he is active at Fraunhofer IPA in Stuttgart, Germany, as project manager for European and national research projects with a focus on software development for robotics. He is currently coordinating the German SeRoNet project, which is tasked to push model-driven software development concepts from the European RobMoSys project into the robotics market. Since 2016 he is a leading member of the ISO TC299/WG6 on Modularity for service robots. His main research interests are on efficient software development methods for reliable software for robotics and AI.

Dr. Mirko Bordignon received B.S. and M.S. degrees in Electrical and Computer Engineering from the University of Padova, Italy, and a PhD in Robotics from the University of Southern Denmark, with research stays at Örebro University, Sweden, and Harvard University, USA. He explored topics in embedded robotics, model-driven software development, and domain-specific languages (DSLs). He then spent 4 years as a software engineer in end-of-line industrial automation, leveraging DSLs and other language engineering techniques to streamline development of .NET and PLC production code deployed in manufacturing environments. Dr. Bordignon joined Fraunhofer IPA in Stuttgart, Germany, in 2015. First as a research scientist and then as a group manager, he worked on software engineering and system integration for service and industrial robots, both within publicly funded research projects and private R&D contracts. A recurring theme championed by his group was open-source robotics software, with specific emphasis on the Robot Operating System (ROS) and its adoption within industrial automation. Since 2019, he is with Google Germany GmbH.