Bootstrapping MDE Development from ROS Manual Code – Part 1: Metamodeling

Nadia Hammoudeh Garcia, Mathias Lüdtke, Sitar Kortik, Björn Kahl, Mirko Bordignon
Fraunhofer Institute for Manufacturing Engineering and Automation IPA
Nobelstr. 12, 70569 Stuttgart - Germany
{nadia.hammoudeh.garcia, mathias.luedtke, sitar.kortik, bjoern.kahl, mirko.bordignon}@ipa.fraunhofer.de

Abstract—**Ten years after its first release, the Robot Operating System (ROS) is arguably the most popular software framework used to program robots. It achieved such status despite its shortcomings compared to alternatives similarly centered on manual programming and, perhaps surprisingly, to model-driven engineering (MDE) approaches. Based on our experience as users and developers of both ROS and MDE tools, we identified possible ways to leverage the accessibility of ROS and its large software ecosystem, while providing quality assurance measures through selected MDE techniques. After describing our vision on how to combine MDE and manually written code, we present the first technical contribution in this pursuit: a family of three metamodels to respectively model ROS nodes, communication interfaces, and systems composed from subsystems. Such metamodels can be used, through the accompanying Eclipse-based tooling made publicly available, to model ROS systems of arbitrary complexity and generate with correctness guarantees the software artifacts for their composition and deployment. Furthermore, they account for specifications on these aspects by the Object Management Group (OMG), in order to be amenable to hybrid systems coupling ROS and other frameworks. We also report on our experience with a large and complex corpus of ROS software used in a commercially deployed robot (the Care-O-bot 4), to explain the rationale of the presented work, including the shortcomings of standard ROS tools and of previous efforts on ROS modeling.**

Keywords—**ROS, models, development environments.**

I. INTRODUCTION

Since the turn of the century a combination of scientific, technical, and economic factors contributed to major transformations in robotics, both in terms of features (e.g., industrial robots with safe human interaction, autonomous guided vehicles without fixed infrastructure) and of scope (e.g., service robotics and the consumer market). Scientific achievements enable increasingly robust navigation and manipulation in unstructured environments. Technical factors include higher-performance, lower-cost computation, communication, sensing units, also thanks to consumer electronics and its economies of scale. Socio-economical trends such as collaborative engineering and new actors entering the field also contribute to the transformation: think of open-source, made even more popular in the last decade by online collaboration platforms; and of the influx of well-funded robotics startups [1]. Notably, many of the new developments are centered on algorithms (e.g., in perception) or aim to improve human robot interaction or reconfigurability (e.g., through user interfaces). They are thus implemented in software, which becomes an enabler as much as a potential limiting factor for the advancement of robotics.

It should not surprise that the combination of such factors originated new software frameworks for robot programming. ROS, one of them, was created in the late 2000s at a well-funded startup (Willow Garage) to develop a robotics platform (the PR2) targeting a nascent market (service robotics), while creating a worldwide community through a platform facilitating open-source collaboration (GitHub). What is surprising, and motivates this work, is that such framework grew into today's arguably most popular and widely used one, while not being superior to others in terms of technical features, nor benefiting from a large development team in the traditional sense. Open Robotics, its steward organization since 2012, consistently reports less than 10 staff members tasked to work on ROS. And yet, a majority of service robotics platforms in the market or under development are ROS-based, with manufacturers of industrial robots also actively exploring this option (ABB [2] [3] and DENSO [4] [3], among the others).

On the other hand, technically superior tools leveraging MDE hold the potential for a step change in software quality by decoupling the description of components and their interactions (the *what*), from their implementation (the *how*), which shifts from a manual, error-prone endeavor to a tool-assisted operation. However, despite their significant uptake in domains such as avionics or automotive, model-based approaches to software engineering have not (yet) become commonplace in robotics, at least for application-level software (except perhaps in niche segments like surgical or space robotics). A thorough examination of the business reasons for this is outside the scope of this paper, although important and worth exploring (size of market vs cost of MDE tools, standard vs vendor-specific programming interfaces, etc). In Sec. II, we will instead share some insights from our practical experience working with industrial customers and building and piloting MDE robotics workbenches, identifying the limiting factors so far preventing us, and possibly other developers of robotics software, to fully embrace MDE. In Sec. III we will review relevant work, to then outline our goal to combine manual programming of ROS systems with MDE benefits in Sec. IV. Sec. V introduces the main technical contribution of this paper, i.e., three metamodels encapsulating ROS nodes, communication patterns, and systems, representing the first step towards our goal. We use a real example in Sec. VI, motivating the development of the tools and offering a concrete use case, and last, we conclude with directions for future work in Sec. VII.

II. PROBLEM STATEMENT

ROS was designed with an emphasis on flexibility of software development: language-neutral, and with a microservices-like architecture for both the runtime and the support tools. A recurring description is that of "simple plumbing", i.e., a middleware offering basic communication primitives (*topics* and *services*) to loosely coordinated units of computation (*nodes*) exchanging data through language-independent data structures (*messages*). None of these ideas were new at the time of its debut, and in fact most scientific publications mentioning ROS are not describing the system but rather reporting it as the platform of choice to run the experiments (on real or simulated hardware) and to share the data to e.g., compare SLAM algorithms. In summary, ROS:

- puts minimal architectural and language constraints, resulting in a low threshold in terms of required expertise. This makes it accessible to most developers but truly mastered by few, since the tradeoff of such flexibility is that unintended software behaviors can easily go undetected;
- is unrivaled in terms of available features (albeit of varying quality), given the ease to create and share ROS packages, due in turn to the low entry threshold. This makes it the easiest and fastest platform to bootstrap a robot platform and/or application with, but the hardest with which to perfect it to a higher level of robustness.

As a testament to how quickly these two aspects made ROS grow, there are now over 3000 available ROS packages covering most robotics subdomains [5], [6]. Our experience with MDE tools, especially when building the ReApp toolchain [7] and using it with system integrators for realistic industrial use cases [8], showed us that they have almost opposite –perhaps complementary?– strengths and weaknesses:

- they force developers to express their designs in terms of models and within a specific development environment. This requires getting familiar with the tooling (a higher entry barrier compared to using established languages), which on the other hand is much less likely to allow for unintended behaviors in the final software artifact;
- much fewer reusable components are available, as so far no single tool bootstrapped an ecosystem comparable to ROS. This entails more development for a new platform or application. However, the software quality of the final artifact will be possibly much beyond that of a prototype.

The two approaches present a seemingly irreconcilable dichotomy. An "early victory" through ROS and its abundance of developers and readily available packages, with the prospect of a much harder path to refined, higher-quality systems. Or, a high "upfront investment" to get familiar with the MDE tool of choice and possibly have it extended to cater for your use case, but with the guaranteed return of a correct-by-construction final outcome. We look for a possible synthesis between these two approaches: leveraging the ROS ecosystem and its ease of use, while using MDE techniques when applicable and worth the investment, so to possibly achieve fast prototyping with the prospect of a tool-supported refinement phase.

III. RELATED WORK

A. ROS and software quality

Given its increasing adoption in commercial applications and other domains demanding production-level code, quality assurance (QA) of ROS software has become the focus for a number of efforts from multiple organizations. QA initiatives focusing on the community development process, as described in [9], are not relevant for the sake of this discussion at this stage. More relevant are techniques and supporting tools concretely inspecting software artifacts, during development (in a Continuous Integration fashion) or afterwards (e.g., audits for something akin to a quality certificate). With regards to testing of individual components, a representative selection of techniques is already adopted by, or at least available to, ROS practitioners. Such techniques range from traditional unit testing [10], to static checking for conformance to style guides [11], to more recent, early developments in applying fuzzing techniques [12]. As these examples indicate, there is ample room for established software QA techniques to be applied to the ROS use case. Tools already exist or are emerging, and their uptake in standard development practices for widely used ROS components will make (or not) the difference and show whether they can significantly impact the software quality of ROS.

With regards to the quality of larger systems composed of nodes or, in turn, of other subsystems, existing software engineering techniques are less directly of help. This since in order to verify system composition, some specific knowledge of ROS internals and conventions is necessary, and so far very few tools embed it. A rudimentary way to verify the correct composition of a ROS system is to use the stock *rostopic* and *rosgraph* tools on a running system. They provide respectively information on active topics (i.e., messages being published by nodes over a named channel) and on the computation graph (i.e., how nodes are composed by establishing said communication channels). The fundamental limitations of these tools are that they work only at runtime, resulting in cumbersome trial-and-error runs, often on real hardware, to test systems for composition; and that they support only topics and not services (i.e., RPC-like interfacing between a client and a server node). One notable, recent example of a tool inspecting the computation graph statically is detailed in [13]. By inspecting API calls in the Abstract Syntax Tree (AST) of ROS code, and by applying remap rules parsed from launch files (i.e., possible renaming of topics and services), the authors seemingly succeed to statically extract a computation graph by means of static analysis. However, they do not apparently perform soundness checks on the graph per se (e.g., to verify "dangling" publishers or subscribers), but rather use it as an intermediate step to then perform property-based testing.

Not surprisingly, to the best of our knowledge the existing approaches from the ROS community all seem to address software QA through a-posteriori checks, rather than by providing some means during development for correct-by-construction composition.

B. MDE efforts targeting ROS

A number of efforts applied MDE to the robotics domain, with differences in terms of metamodels (e.g., UML or specialized ones such as RobotML [14]), targeted architectures (e.g., Urbi [15], OROCOS [16], ROS, simulation engines), and support for specific paradigms such as separation of concerns or features such as generation of memory-efficient code. A characteristic often shared is the use of Eclipse tools, specifically the Eclipse Modeling Framework (EMF) and its ecore implementation of the OMG eMOF (essential Meta Object Facility) specification. Tools built on Eclipse are mentioned due to our previous experience with them as users or developers, which informed our current line of work.

SmartSoft [17] has a service-oriented component-based approach covering the entire robot development process by defining the components skeleton (a wrapper for the user code), the interfaces and patterns for their communication. It uses EMF, Xtext and Sirius as implementation technologies, a choice which we decided to follow. Its latest version offers advanced features such as runtime behavior simulation and performance analysis [18], with the understandable drawback of not allowing import of existing components originating outside of SmartSoft. The RobMoSys project [19] leverages SmartSoft and ROS, among the others, and promises predictable composition of both existing and new components through MDE. The BRICS (Best Practices in Robotics) [20] component model (BCM) combines 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 [21]), developed at our organization Fraunhofer IPA, bridges 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" [21]. BRIDE showed how to fit ROS in a model-based approach, and the experience with it and with the following project ReApp [22], together with the experience of manually developing large robot systems with ROS, motivates the work presented in this paper.

Compared to the top-down approach of BRIDE and the ReApp Workbench, both aiming for system development to happen within their IDEs with ROS serving merely as the backend architecture, we aim to address:

- the import through modeling of existing, manually developed ROS systems, so as to make them first-class citizens of the MDE tooling;
- verification of system properties by exploiting such modeling, usable to check for constraints typically verifiable only at runtime (e.g., "is a publisher for a topic that this node is subscribing to present in the system, and if so, does it use the same message format?");
- interfacing with other systems, similarly encapsulated in a model exposing their basic communication mechanism.

Some shortcomings of BRIDE, due to its different goals, made us decide to reboot our MDE-ROS effort, rather than continuing to develop it. The most significant of them are the single ecore model (ROS DSL) conflating several distinct concepts, and its lack of a generic (non ROS) component interface, needed to encapsulate subsystems and/or to allow hybrid systems.

IV. VISION: BOOTSTRAPPING MDE DEVELOPMENT WITH ROS MANUAL CODE

Our vision is motivated by a belief rooted within our group: in the wide and varied robotics domain, manual code, meaning *software developed using manual processes*, is not going to be completely superseded in the mid-term. Reasons for this are:

- an all-encompassing MDE tool, catering for the myriad of use cases and hardware platforms, and also gathering the community needed to co-create or otherwise economically sustain a viable ecosystem, is theoretically possible, but unlikely to be achieved in practical terms;
- most current robots are built on manual code, which is to be dealt with for the foreseeable future;
- for fast prototyping manual coding is likely to continue being the best option, given the cost of modifying the tools to accommodate for unforeseen use cases.

On the other hand, the advantages of MDE are evident to us, after years of experiencing firsthand the problems of manual coding. We envision the ideal MDE tool not as a "strait-jacket" for the developer, where little can go wrong at the cost of operational inflexibility (no manual code, nothing foreign to the model can be imported, etc), but rather as a "scaffold", i.e., providing support on-demand and non-obtrusively to the manual code bootstrapping the development process. A way to do this is to model manual code artifacts, in order to make explicit any convention or assumption otherwise implicit. In this spirit, the ROS metamodels presented in the next section allow users to describe, with minimal models, manually developed ROS systems, and thus to capture their interaction patterns through their external communication interfaces. Such model overlay can potentially be used to:

- *improve the understanding* of what will happen at runtime. E.g., will a publisher and a subscriber correctly communicate, or does an uncommon message format break some (implicit) assumptions?
- *ease interaction* between ROS and other systems. A model focused on the communication and the semantic annotation of its components and interaction patterns can facilitate M2M transformations with other middlewares and their communication artifacts;
- *diffuse best practices* on manually written code. A database of models for representative ROS components and their interaction patterns could let an IDE highlight inconsistencies, IntelliSense-style: "camera component using non-standard communication patterns and message formats: is this intended?". Such database could be populated through static code analysis (HAROS is for instance already capable to extract some interaction patterns [13]).

V. METAMODELING - FROM ROS TO SYSTEMS

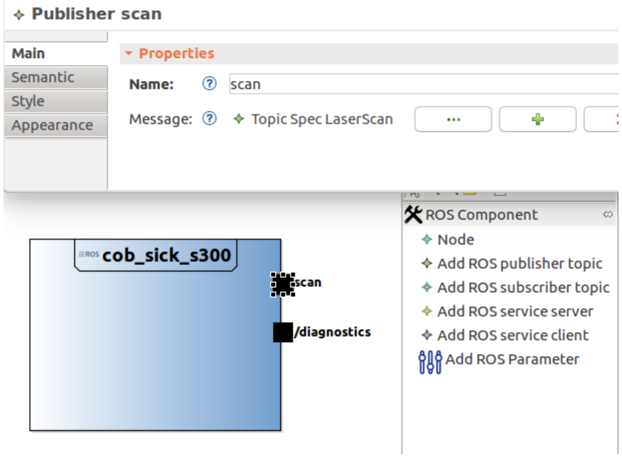
Our metamodeling effort starts by reviewing ROS basic concepts, in order to define a simple ROS metamodel to capture them. We then introduce the Component Interface metamodel, used to lift ROS specific details and raise the abstraction level to generic components modeled after OMG standards, to cater also for the modeling of non-ROS systems if necessary. Last, the System metamodel is introduced to describe the composition of Component Interfaces.

This family of three metamodels corresponds to three matching Xtext DSLs, which allow the user to check and validate against properties and constraints, as described later. This kind of checks, although simple in the SICK laser scanner component used as an example, is a proof-of-concept usable in the larger Care-O-bot description of the next section. Our experience suggests that when scaled to such larger system, the convenience will become evident and significant. All the code for this work, including the example, is publicly available at https://github.com/ipa-nhg/ros-model.

A. ROS metamodel

ROS is conventionally seen under three "lenses": the filesystem (how code is organized and stored), the computation graph (how systems are split in processes and how these interact), and the deployment mechanism (how entities are distributed, named, accessed at runtime). Distilling the basic constituent entities from the system, we look once again to the "simple plumbing" analogy of ROS as a peer-to-peer network of processes (*nodes*) exchanging data. The communication mechanisms are the *topic* and *service* patterns. Topics are means for one-way communication and many-to-many connections, while services are used for request-response communication and one-to-one connection. Finally, the objects of this communication are *messages* definition for *topics* and the *services* definition for *services*, which consist in language-independent data structures composed of primitive data types (String, Double, Int, Boolean, etc). Our ecore ROS metamodel describes all these core concepts. Fig. 1 shows a screenshot of our tool describing graphically the model of a pre-existing ROS driver for a SICK scanner [23] and an extended properties view of the *scan* publisher. By layering such minimal model on manually developed code artifacts, we can model also their interaction patterns by describing the communication interfaces. Such interfaces are defined by 1) the number of participants (one-to-one or many-to-many), and the direction of the flow of information (input or output) and 2) the object type of the communication (the data structure of the messages being exchanged).

As a first example of how we can leverage this simple model, the Xtext [24] plugin defining the grammar of the DSL corresponding to this metamodel has access to a library of ROS standard messages and services, collected from the ROS wiki documentation. Code in Lst. 1 shows an example of the validated model output of the model represented in Fig. 1(the file *sick_s300.ros*).



Name: scan

Message: Topic Spec LaserScan

Fig. 1. Modeling example for a ROS driver for a the Sick scanner s300

ROS naming conventions verified by the Xtext plugin include rules like those defined in REP 144 [25] for ROS package names: a name must only consist of lowercase alphanumerics and "_" separators.

```
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 "/diagnostics"
          message "diagnostic_msgs.DiagnosticArray"
  }}}}
}}}}
```

Listing 1. sick_s300.ros file, the Xtext format of the node represented in Fig. 1

B. Component Interface metamodel

As ROS concepts have been captured, and verification steps on the information provided by the user have been performed (message types, naming conventions), we now lift these ROS specific language concepts to allow for communication between different frameworks. For this purpose we evaluated the OMG specification "Deployment and Configuration of Component-based Distributed Applications" [26] and adapted the ROS model to a component-based structure. The main concept of a component-based architecture is to partition the applications into small, possibly reusable components that can interact with each other through input and output ports. An advantage of this concept is its recursive nature, which allows the encapsulation of a set of interconnected components as a component by itself. This OMG standard defines an application as "a component that is assumed to be independently useful...this component can be implemented directly (by a monolithic implementation), or it can be implemented by an assembly, where the implementations for its subcomponents can again be either monolithic or assemblies". Adhering to this guideline, we defined a metamodel that, based on the ROS metamodel just described, adapts it to the concept of Component Interface. That is, according to the OMG standard, "a named set of

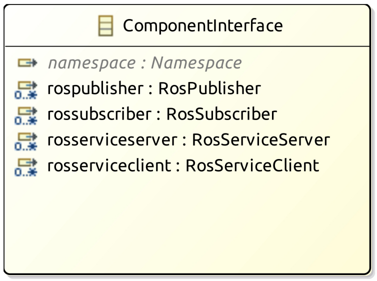

ComponentInterface
namespace : Namespace
0..*
rospublisher : RosPublisher
0..*
rossubscriber : RosSubscriber
0..*
rosserviceserver : RosServiceServer
0..*
rosserviceclient : RosServiceClient

Fig. 2. Class model for a component interface model

provided and required interfaces that characterize the behavior of a component”. The interfaces that ROS requires to allow interconnections are the communication mechanisms known as topics and services. Partitioning inputs and outputs, for topics this results in: *publishers* and *subscribers*; and for services in: *servers* and *clients*.

As previously mentioned, the third lens to see ROS through is the deployment mechanisms, i.e., how computational entities are distributed, named, accessed during runtime execution. To group entities (“components”, in the parlance of this subsection) ROS provides a hierarchical naming structure used for all resources in a ROS Computation Graph, where each resource (node, topic, etc) is defined within a namespace. Resources can create further resources within or above their own namespace, and the connections can be made between resources in distinct namespaces, as is often the case for systems resulting from large scale composition.

Fig. 2 shows the ecore implementation of the Component Interface structure for ROS. In it we reference topic and service definitions from our ROS specific model, preserving the definition of the nature of the interfaces, and add the definition of namespaces for the entire component and for each interface. By doing so, we maintain the flexibility of the ROS deployment mechanism, allowing the definition of global, relative and private namespaces for interfaces within the same node. At the same time this provision addresses adequately the OMG definition of component interface as a set of interconnected components.

Code in Lst. 2 shows an example of the Component Interface transformation of the model of Fig. 1, where:

- the ROS node *cob_sick_s300* is allocated under the namespace *base_laser_front*;
- the publisher of the scanner output data is remapped to this namespace;
- the standard topic */diagnostics* stays as global topic.

The transformation by adaptation of ROS code to a generic standard concept such as Component Interface enables the use of the same model to describe monolithic ROS nodes and large ROS systems, achieving the two main objectives of 1) simplifying the deployment process of ROS systems by defining a system as a composition of sub-systems and 2) facilitating the interoperability of an entire ROS application with other component-based architecture frameworks.

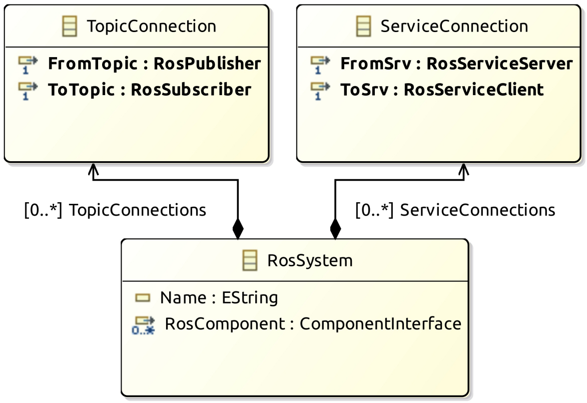


Fig. 3. Class model for the *rossystem* model

```
ComponentInterface {
 name base_laser_front
 NameSpace RelativeNamespace {parts{”/base_laser_front” }}
 RosPublishers{
  RosPublisher ”/base_laser_front/scan” { ns ”/base_laser_front”
  RefPublisher ”cob_sick_s300.cob_sick_s300.scan” },
  RosPublisher ”/diagnostics” {
  RefPublisher ”cob_sick_s300.cob_sick_s300./diagnostics”}}
}
```

Listing 2. Example of a component interface based on the sick_s300_node given in Fig. 1

C. System metamodel

ROS systems can be grouped and started through launch files, an XML structure used to specify both standalone applications or subsystems. The launch files define which nodes will be launched, from which package, with which arguments and in which namespaces. Nodes could be grouped within a namespace and/or within the machine where they are started. Another powerful ROS feature, embedded as a tag for the launch files structure, is the remapping, which allows to pass complex name assignments.

Currently launch files need to be written manually, with the developer responsible for checking the proper definition of the interconnections between nodes. The lack of validation techniques for such communication details at design-time is, in our experience, one of the biggest shortcomings of ROS, making it hard to verify that applications behave as intended. It also represents tedious work of implementation and debugging for systems integrators, as the next section will illustrate on a representative example from a commercially deployed robot which our team worked on.

Based on this observation, our third metamodeling tool makes use of Component Interface models to compose ROS nodes, sub-systems and systems, for which we can validate at design-time the respective interconnections. The representation of our ecore model to describe ROS systems is shown in Fig. 3. We also accounted for the OMG specification “Deployment and Configuration of Component-based Distributed Applications” [26] of a connection defined as “either a communication path among the ports of two or more subcomponents allowing them to communicate with each other, or (it is) a communication path between an assembly's external ports and an assembly's subcomponents that delegates the external ports behavior to the subcomponents ports". In our case, *TopicConnections* and *ServiceConnections*.

To validate the composability of nodes we identified the two main causes of a mismatched communication among ROS processes:

- *Disparity of the communication object*, i.e., the subscriber of a topic asks for a different message type than the one being published under the same topic name. In this case the *rosmaster* will show a warning message and do not allow the transfer of information.
- *Disparity name of the communication agents*, i.e., the name of the subscriber does not match the name of the expected publisher. This case is critical because for ROS, from the point of view of the architecture, this does not represent an error and will not result in any warning, not even during execution.

Our tool evaluates these two rules and emits an error at design-time if the user tries to join two interfaces with different names or mismatched messages types. This implies that the model of the system developed using our tools will undergo a validation step before being generated.

Once the interconnection of different ROS nodes is validated, we can also automatically generate the deployment artifacts code (i.e. *roslaunch*, *rosinstall* and installation scripts files), reducing considerably the effort and time for integration, preventing typing errors, and ensuring that the generated files have been checked against mismatched communication. Currently, launch files are generated, with rosinstall representing a simple addition under development to be made available soon.

Fig. 4 shows one example of a ROS system defined using our graphical tool. This particular diagram represents a node combining and unifying the outputs of three different scanners, all based on the same ROS node, i.e., the model of Fig. 1, and mapped to different namespaces using our component interface model. It also integrates the standard *diagnostics_aggregator* node, with all connections having undergone automatic check and validation.

VI. A MOTIVATING EXAMPLE: THE CARE-O-BOT 4

The main goal motivating the work just presented is to facilitate the composition of large systems, task which our group performs on a regular basis both for internal development and for customer projects. We aimed to leverage lessons learned during the development of the Care-O-bot family of service robots, especially with regards to the improvement of ROS deployment mechanisms for systems. We have been successfully using ROS native systems to control service robots since the Care-O-bot 3 debuted in 2008, and more recently on the Care-O-bot 4, which was also deployed commercially at retail locations throughout Germany. During the development of the Care-O-bot family we suffered from the lack of tools for the design of the software architecture, the autogeneration of deployment artifacts and, more importantly, for the debug, test and introspection of large systems.

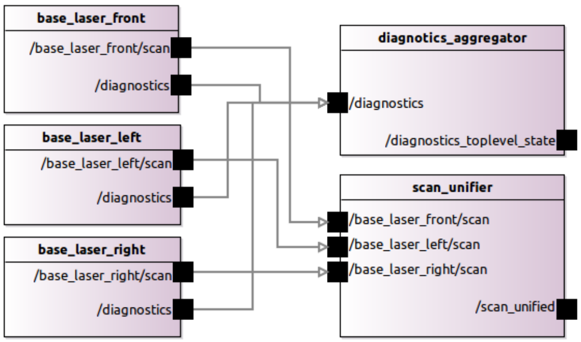


Fig. 4. Graphical representation of the composition of ROS components using the *rossystem* metamodel and its validation tools at design-time

To give some perspective on a concrete and representative case of ROS system for Care-O-bot 4, we can mention that just the drivers to control the hardware (i.e. the lowest-level components) are over 100 ROS nodes. Counting the number of possible interconnections between nodes, the robot manages over 500 topics and about 500 services. In addition to the high number of nodes, topics, and services, another dimension for the complexity so far manually tackled is given by one of the launch files. It recursively includes several levels of launch files redefining namespaces and hierarchies multiple times and ultimately resolving to the actual driver nodes. This huge infrastructure was created manually and can only be validated by starting the entire robot system at runtime (using tools like rosgraph) and testing empirically on the real robot all its functionalities.

The particular case of Care-O-bot gets even more complex given the emphasized modularity of the robot. That is, every module should be able to work independently, but also be composed and exchanged with others. This implies that there are several versions of the same robot in terms of software integration: for each version a new launch file has to be created and tested at runtime on the real hardware. Due to this, and to facilitate the development, debug and test of new robot versions, the software infrastructure has to be divided into separate sub-modules that can be checked independently of each other. The cob_robots [27] repository was created to hold this deployment complexity, containing launch files to start separately (with the correct configuration for each robot version) the drivers of a component and the drivers of a set of component whose composition provides a given functionality. A final, top level XML file contains the composition and distribution on computing machines of the previous launch files and extra tools for the user interaction.

To give the reader an overview of this complexity, Fig. 5 shows a full rosgraph of the nodes and topics running *just* on the base module (a third of the full system) to run the low-level

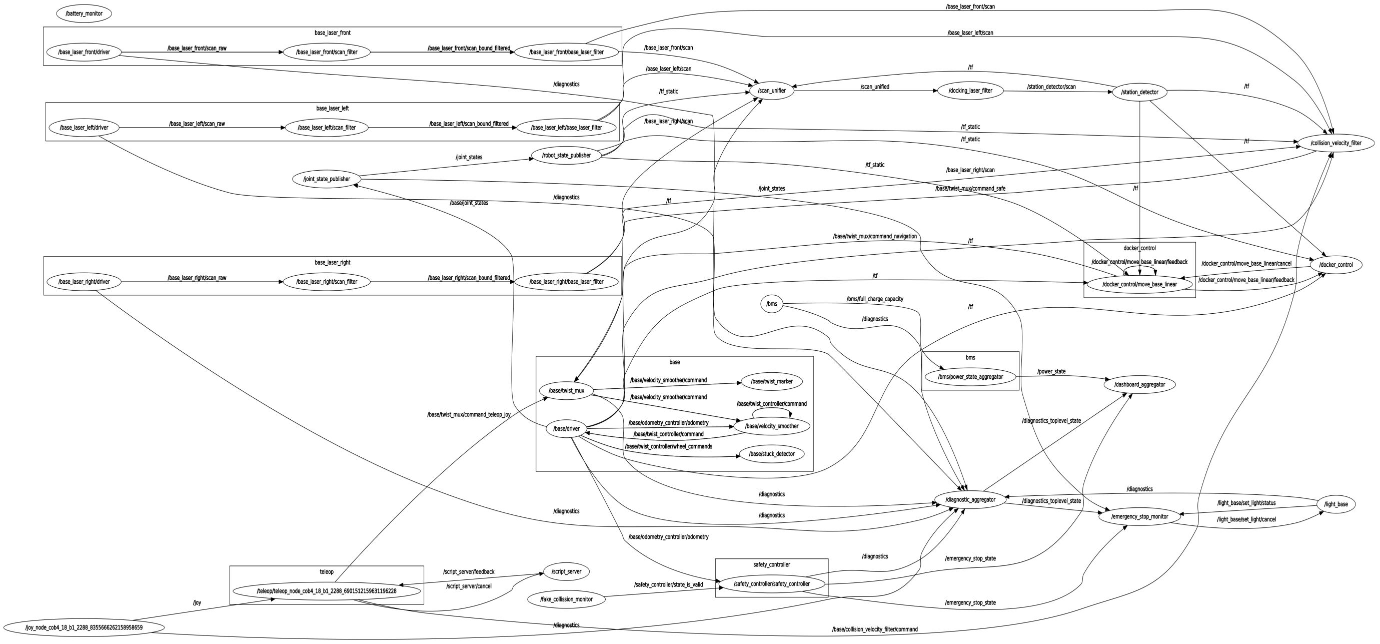

/battery_monitor
base_laser_front
/base_laser_front/driver
/base_laser_front/scan_raw
/base_laser_front/scan_filter
/base_laser_front/scan_bound_filtered
/base_laser_front/base_laser_filter
/base_laser_front/scan
/base_laser_left/scan
/diagnostics
/tf_static
/scan_unifier
/tf
/scan_unified
/docking_laser_filter
/station_detector/scan
/station_detector
base_laser_left
/base_laser_left/driver
/base_laser_left/scan_raw
/base_laser_left/scan_filter
/base_laser_left/scan_bound_filtered
/base_laser_left/base_laser_filter
/base_laser_right/scan
/tf_static
/tf_static
/collision_velocity_filter
/joint_states
/robot_state_publisher
/joint_state_publisher
/tf_static
/base_laser_right/scan
/joint_states
/base/twist_mux/command_safe
/diagnostics
/base/joint_states
/tf
/base/twist_mux/command_navigation
/tf
docker_control
/docker_control/move_base_linear/feedback
/docker_control/move_base_linear
/docker_control/move_base_linear/cancel
/docker_control/move_base_linear/feedback
/docker_control
base_laser_right
/base_laser_right/driver
/base_laser_right/scan_raw
/base_laser_right/scan_filter
/base_laser_right/scan_bound_filtered
/base_laser_right/base_laser_filter
/tf
/bms
/bms/full_charge_capacity
/diagnostics
/tf
bms
/bms/power_state_aggregator
/power_state
/dashboard_aggregator
base
/base/twist_mux
/base/velocity_smoother/command
/base/twist_marker
/base/velocity_smoother/command
/base/twist_controller/command
/base/odometry_controller/odometry
/base/velocity_smoother
/base/driver
/base/twist_controller/command
/base/twist_controller/wheel_commands
/base/stuck_detector
/base/twist_mux/command_teleop_joy
/diagnostics_toplevel_state
/diagnostics
/diagnostic_aggregator
/diagnostics_toplevel_state
/diagnostics
/diagnostics
/light_base/set_light/status
/light_base
/emergency_stop_monitor
/light_base/set_light/cancel
/base/odometry_controller/odometry
/diagnostics
safety_controller
/emergency_stop_state
teleop
/script_server/feedback
/script_server
/script_server/cancel
/teleop/teleop_node_cob4_18_b1_2288_6901512159631196228
/safety_controller/state_is_valid
/fake_collission_monitor
/safety_controller/safety_controller
/joy
/emergency_stop_state
/joy_node_cob4_18_b1_2288_8355666262158958659
/diagnostics
/base/collision_velocity_filter/command

Fig. 5. Care-O-bot4-10 graph nodes example of the base module

hardware drivers. This example pertains to version 4-10 of the Care-O-bot robot. Understandably, once integrating necessary components like navigation, manipulation and perception, the complexity to be dealt with increases even more.

Using tools such as those proposed in this paper, the deployment phase for new versions of Care-O-bot combining already existing component could benefit from savings in time and effort, by autogenerating the full deployment software (launch files), and from validation at design time, significantly easing the debug and test phases. They would facilitate even more the design of new applications combining high level functionalities by Care-O-bot architecture experts, as they could create new functionalities by connecting software components using the tooling.

VII. CONCLUSIONS AND FUTURE WORK

The advantages of MDE are crystal clear to us, after years of experiencing firsthand the problems of manual coding in systems software-wise as large and complex as the Care-O-bot. However, past experience with model-to-code generation tools leads us to also believe that a pure MDE approach would not be feasible, due the problem of "bootstrapping" both the development environment and commercially deployable applications from models at the same time. Having to produce, maintain, and work with manual code, colloquially meaning *software developed using manual processes*, is for us and probably for many robotics practitioners an established fact that will be relevant still for years to come. Our vision is to combine the two approaches by leveraging manual code when available, and modeling it in order to make conventions and assumptions explicit and possibly verifiable.

In this paper, we presented a set of metamodeling tools, already implemented (albeit in prototypical stage) and made available as open-source. The tools allow users to describe software artifacts, both preexisting and fictitious, starting from concrete ROS nodes up to the composition of implementation-agnostic subsystems. Practical advantages of the metamodeling tools and their DSL counterparts is a number of assistive actions to validate the composition of systems (messaging structures and patterns) and to execute their deployment (automatic generation of launch files). A motivating example from a real, commercially deployed, robot, shows the high relevance to robot practitioners of our tools.

We plan to continue our work in this spirit, leveraging our experience from previous projects to expand the results presented in the paper to a collection of *descriptive* and *prescriptive* tools. Descriptive, since they would serve as an atlas to map commonly used interactions patterns of existing components. Prescriptive, since they would highlight anomalies and possibly suggest interventions. Two considerations encourage us to pursue this course of action. First, the ROS metamodel is intentionally minimal to be applied to any existing ROS artifact, and non-obtrusive so as not to necessitate of e.g., special coding conventions. Second, populating a model database to identify possible deviations from commonly used message and interaction patterns is a task arguably amenable to automation, e.g., through static analysis tools [11] and even runtime monitoring scripts. In parallel to the extension of the tools, we intend to consolidate them by testing on larger examples such as the illustrated subset of the Care-O-bot. Acknowledging the multiplicative power of open-source, finding new users for the tools will also provide new inputs for other directions of our work.

However, two other considerations make us aware of possible open matters. First, due to the heterogeneity and highly dynamic nature of the robotics domain, can a model-based approach keep up pace and grow as fast as the robotics world does? Second, this approach attempts to connect two typically disjoint profiles of developers: the enthusiasts of hand-written code, and the proponents of structures and encapsulation of software in pre-defined formal concepts. How likely is it for work like the one presented here to find adopters among these two communities?

We aim to work on these open matters in the scope of the Service Robotics Network (SeRoNet) project [28], which explicitly envisions multi-platform hybrid architectures (both hand-written and model-based) and tooling support to the developers. Also, as part of our contribution to the SeRoNet project the work exposed in this paper will be maintained, improved and augmented in terms of features, among which is a more comprehensive deployment system. To achieve it, in addition to the generation of launch files of the final system model we target the creation of the rosinstall files and the scripts for rosdep and other ROS tools needed to automatically deploy full ROS systems. As part of a future contribution we are also evaluating the use of mechanisms for the verification of applications at task planning level.

The work and the example presented in this paper are publicly available on GitHub https://github.com/ipa-nhg/ros-model/.

ACKNOWLEDGMENT

The research and development project that forms the basis for this report is funded within the scope of the "PAiCE Digitale Technologien für die Wirtschaft" technology programme run by the German Federal Ministry for Economic Affairs and Energy and is managed by the DLR project management agency "Gesellschaft, Innovation, Technologie - Informationstechnologien/Elektromobilität" at the German Aerospace Center in Cologne. The author is responsible for the content of this publication.

REFERENCES

[1] "The Robot Report: Startup Companies," Nov. 2018. [Online]. Available: https://www.therobotreport.com/map/start-up-companies/

[2] "Drivers by ABB for EGM and RWS interfaces to its robots." [Online]. Available: https://github.com/ros-industrial/abb_libegm, https://github.com/ros-industrial/abb_librws

[3] "ROS-Industrial Conference 2018 - program and proceedings." [Online]. Available: https://rosindustrial.org/events/2018/12/11/ros-industrial-conference-2018

[4] "Drivers by DENSO for its VS060 and Cobotta robots." [Online]. Available: https://github.com/DENSORobot

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

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

[7] Y. Hua, S. Zander, M. Bordignon, and B. Hein, "From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning," in *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ETFA.2016.7733579

[8] "ReApp project - demonstrators." [Online]. Available: http://www.reapp-projekt.de/index.php?id=pilot_demonstrators

[9] A. Alami, Y. Dittrich, and A. Wasowski, "Influencers of quality assurance in an open source community," in *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2018, pp. 61–68. [Online]. Available: http://doi.acm.org/10.1145/3195836.3195853

[10] "ROS and quality: Automatic Testing with ROS," http://wiki.ros.org/action/show/Quality/Tutorials/UnitTesting, accessed: 2018-07-20.

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

[12] Z. Fu, "Quality Assurance Initiatives for ROS - Tool Development," https://roscon.ros.org/2018/presentations/ROSCon2018_ROSIN2.pdf, 29 September 2018.

[13] A. Santos, A. Cunha, and N. Macedo, "Property-based testing for the robot operating system," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2018, pp. 56–62.

[14] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 149–160.

[15] J. . Baillie, "URBI: towards a universal robotic low-level programming language," in *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Aug 2005, pp. 820–825.

[16] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, May 2001, pp. 2523–2528 vol.3.

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

[18] "SmartSoft V3 Technology Preview." [Online]. Available: http://www.servicerobotik-ulm.de/drupal/?q=node/83

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

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

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

[22] R. Awad, G. Heppner, A. Roennau, and M. Bordignon, "ROS engineering workbench based on semantically enriched app models for improved reusability," in *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–9.

[23] "Documentation of the ROS driver for the SICK visionary." [Online]. Available: http://wiki.ros.org/cob_sick_s300

[24] "Xtext project website information." [Online]. Available: https://www.eclipse.org/Xtext/

[25] "REP 144: ROS Package Naming." [Online]. Available: http://www.ros.org/reps/rep-0144.html

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

[27] "Care-O-bot deployment repository: cob_robots." [Online]. Available: https://github.com/ipa320/cob_robots

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