A use case in model-based robot development using AADL and ROS

Gianluca Bardaro
Politecnico di Milano
Milan, Italy
gianluca.bardaro@polimi.it

Andrea Semprebon
Politecnico di Milano
Milan, Italy
andrea.semprebon@mail.polimi.it

Matteo Matteucci
Politecnico di Milano
Milan, Italy
matteo.matteucci@polimi.it

ABSTRACT

Designing a robotic application is a challenging task. It requires a vertical expertise spanning various fields, starting from hardware and low-level communication to high-level architectural solution for distributed applications. Today a single expert cannot undertake the entire effort of creating a robust and reliable robotic application. The current landscape of robotics middlewares, ROS in primis, does not offer a solution for this problem yet; developers are expected to be both architectural designers and domain experts. In our previous works we used the Architecture Analysis and Description Language to define a model-based approach for robot development, in an effort to separate the competences of software engineers and robotics experts, and to simplify the merge of software artifacts created by the two categories of developers. In this work we present a practical use-case, i.e., an autonomous wheelchair, and how we used a combination of model-based developed and automatic code generation to completely re-design and re-implement an existing architecture originally written by hand.

KEYWORDS

model-based, robotics, AADL, ROS, code generation

1 INTRODUCTION

Robots are complex systems: sophisticated and heterogeneous hardware components coordinated by a software architecture which structures the execution of advanced algorithms. Robots are the result of the cooperation of experts from various fields: mechanical engineering, electronics, control theory, software engineering, artificial intelligence, and more, depending on the application. Often these experts interact with each other on the common field of the software architecture, each one providing a small implementation to solve a domain specific problem, this could be either a device driver, a low-level control system, a high-level planner or the underlying communication infrastructure. However, no real support for this cooperation exists which separates the efforts of each expert and, at the same time, combines them seamlessly.

Robot software architectures are typically designed as distributed component-based systems (see [4] for a survey), most of them implemented exploiting the infrastructure provided by a robotic middleware [13]. Currently, the most used is the Robot Operating System (ROS) [15]. It is based on a system composed by nodes and topics. Nodes are processes and each implements a precise functionality; this may be more or less complex depending on the case, e.g., a device driver or a control loop. Topics are named communication channels used to exchange messages between nodes in a publish/subscribe fashion. In this scenario one would expect ROS to lay a specific structure a developer can use to implement the nodes, but this is not the case.

One of the reasons of ROS early popularity was the simplicity of its implementation and the freedom left to the developers, to cite ROS designers motto: *"we don't wrap your main"*. While this unlimited freedom, and lack of structure, was viable almost ten years ago when robotic applications were mostly prototypes and small academic implementations, today this is not sustainable. Indeed, efforts existed already to adopt a more formal approach to robot development. For example OROCOS [5] tried to introduce *design good practices*, RoCK [12] proposed a system description based on Ruby, and SmartSoft [16] introduced a model-driven development approach. None of them has yet reached a level of diffusion comparable to ROS, therefore they end up abandoned or they are used by relatively small research groups.

Other efforts were aimed at the general development process, like BRICS [6], which tried to promote robotic software reuse through software product lines and model-driven development, RobotML [9], an extension of the UML format to describe robotic applications, or RobMoSys, a currently active project aimed at coordinating the robotic community in creating a structured robot ecosystem at the level of robotics models and meta-models. It is possible that RobMoSys will succeed where others have failed, since it recognizes one of the strength that ROS has: the community. Indeed, in the years, hundreds of developers created components and made them available to the public to be used in various projects: a new ROS developer can bootstrap his first robot by writing a small amount of code.

Nevertheless, robotic systems complexity grows dramatically when adding functionalities and working outside already existing components; often developers are unprepared to this growth, especially when they are expert on a specific field without a strong background in software development. A typical example is given by

control experts porting their own control algorithms from a simulated environment to the real robot. These developers not only need an architectural overview of the system, to provide the structure and connections between components, but they also ought tools to help in the design the inner working of each of these components; especially if they need to adapt existing algorithms and libraries.

The long term aim of our work is to exploit a model-based approach to create a description of a robot architecture and use this model to do architectural analysis and automatic code generation. To do so we use the Architecture Analysis and Description Language (AADL) [10] to describe the model (details of this are described in our previous work [2]). We use ROS as our target for code generation, to achieve a double objective: (I) provide a complete toolchain that goes from the model to the implementation and (II) enhance the ROS implementation in a way that will help developer create better nodes and distinguish between framework-related and problem-related implementations.

In this paper we present a use case in model-based design where we took an existing working robot, with an architecture developed by hand by a roboticist, and, starting from that, we developed a new architecture aiming at the same functionalities. The new architecture is entirely based on automatically generated ROS nodes from their abstract description in the Architecture Analysis and Design Language (AADL) as presented in [2]. Section 2 provides a description of the platform used, in Section 3 we describe in details how the software was modeled using AADL. In Section 4 and Section 5 we discuss how we actually implemented the robot architecture, first detailing the structure of our base ROS node, and then describing how we applied this reference design to all the nodes in the architecture. Section 6 compares the outcome of the model-based development against the original architecture, while in Section 7 we discuss our results and present some related works. Section 8 presents some relevant conclusions to be derived from the work.

2 THE ROBOT PLATFORM

The robot used in this work is an electric wheelchair modified to be controlled with a computer, and equipped with a collection of sensors and other hardware components as part of the ALMA project [1]. The underlying wheelchair is a standard model produced by Degonda Rehab SA; it is the *Twist T4 2x2*, suitable for indoor and outdoor use, with great maneuver capabilities thanks to its two-wheeled dynamics. The system which converts the wheelchair in a robotic platform is called Personal Mobility Kit (PMK), and it consists of:

- motor encoders, to provide wheel odometry.
- two *Sick TiM 561* laser scanner distance sensors, to add full environmental perception.
- the *Shuttle DS81L* used as the on-board PC to run ROS.

In principle, the Personal Mobility Kit can be interfaced with any wheelchair control system since it is an add-on mounted over an already existing hardware. In our work we build the interface to communicate with the on-board electronics manufactured by Penny&Giles Drive Technologies Ltd. (PGDT). Extending the PMK to work with other electronic systems would not be overly complex,

since it would require a change of interface (namely software and hardware elements), but not a modification of the core of the PMK in terms of control algorithms.

A standard wheelchair is controlled using a joystick, placed generally on one of the armrests, this provides manual control with no native support for assisted or autonomous movements. The software components of the PMK extends the functionalities of the wheelchair, by implementing the following drive modes:

- **Fully manual with PMK tuned off**, i,e, the wheelchair is controlled via the on-board joystick. This is the native driving mode of the platform, it has to be available even after the modification introduced by the PMK.
- **Fully manual but mediated by PMK**. Movements are controlled by a wireless joypad or by using the on-board joystick.
- **Assisted**, meaning that the movements, requested indifferently by the joystick or the joypad, are processed by the PMK to avoid possible obstacles perceived by the lasers.
- **Fully autonomous**, in which the wheelchair control is taken by the PMK which drives around a known environment (i.e., already mapped) circumventing obstacles to reach a requested goal.

3 THE MODEL

Fig. 1 represents the whole architecture of the system described using AADL, while tools [11] exist to aid AADL development and creation of graphical representation of the textual model, the version presented here is created by hand to increase clarity. We removed some details, namely the unused ports on some nodes (i.e., *move_base*) and the binding of each topic on the corresponding virtual ROS bus.

For the definition of the model we exploited nesting capabilities of the *system* component to organize and generalize the architecture. We divided the main system in four subsystems, each capturing a specific functionality of the robot: teleoperation, sensing, navigation and platform.

3.1 Teleoperation

This subsystem encapsulates all the software and hardware components to implement teleoperation. In this case it includes an *XInput* joypad, specifically a Logitech Gamepad F710, the ROS driver node *joy_node* from package *joy*, which reads input from the hardware and converts it into a ROS message, and the custom node *joy_control* to convert the message from the driver to velocity commands. The subsystem also includes the physical USB connection used by the joypad, modeled as a physical bus. The entire subsystem exposes a single outgoing port associated with the velocity command, this makes it possible to replace the teleoperation module in the model with any other configuration, e.g., a different physical controller, without changing the whole structure of the system.

3.2 Sensing

ROS Navigation requires at least two information sources: a laser range finder and the odometry. The aim of the sensing subsystem is to abstract multiple laser scanners mounted on the robot into a single source of information; we modeled it with a single outgoing port representing a scan topic. This subcomponent contains the
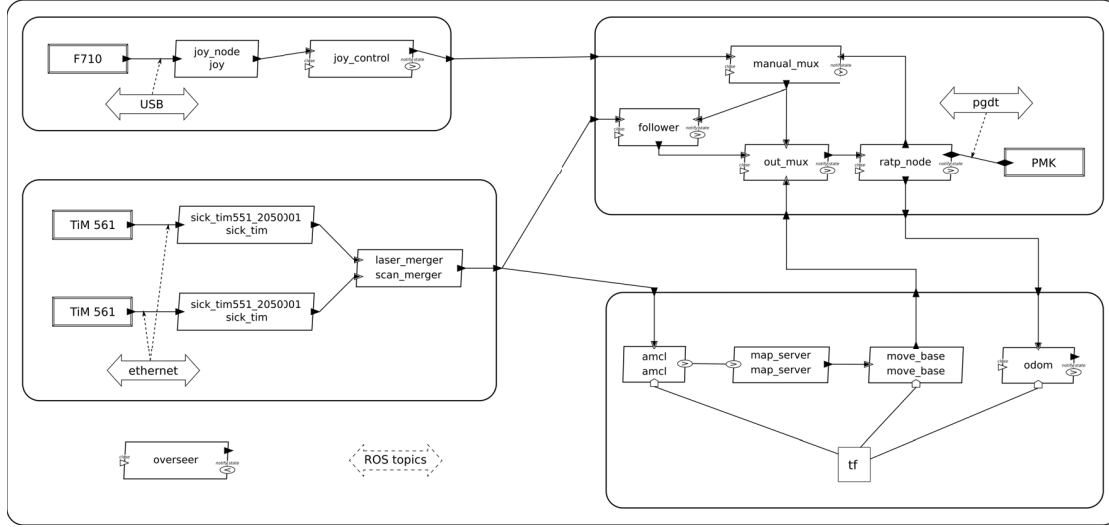
---

[1]http://www.alma-aal.org/

Figure 1: Graphical representation of the AADL description of the whole architecture. The figure show the hierarchical division of the obtained using the *system* components. Some connections are omitted for clarity.

hardware model of the two Sick laser range finders, their respective driver, each one a different instance of the same process model, and a node used to merge the scans from each laser into a single output. Messages coming from this scan merger are then relayed outside the subsystem. As for the teleoperation subsystem, the sensing module includes the physical bus connecting the laser range finder to the system, in this case an ethernet connection.

3.3 Navigation

This subsystem mostly contains legacy nodes from the ROS Navigation stack, with the addition of a custom node used to integrate the speed of the robot and estimate its local position. The data component used to model *tf* is included here as a mean to share the position of the robot. The navigation subsystem receives sensor sources, the robot speed and the scans, to output a velocity command. No hardware component is present, being this a purely software submodule. Abstracting this part of the architecture and clearly defining what kind of input and output the navigation subsystem expects is particularly important given our aim with the autonomous wheelchair. Indeed, we want to use it as a reference platform to test different algorithms and architectural solutions for navigation.

3.4 Platform

All the platform specific nodes and hardware components are contained in this subsystem. A device component models the Personal Mobility Kit used as low-level interface between the wheelchair and the software architecture. This hardware component is connected using a special bus to the custom made driver *ratp_node*, which also works as a bridge between non-ROS data streams and ROS messages. The other nodes are multiplexers for manual or autonomous driving specifically designed for this robot given the peculiar double manual configuration (i.e., driving with the on-board joystick

or with the remote joypad), and a node implementing local obstacle avoidance for assisted driving. The whole subsystem is designed with generality in mind, therefore it exposes three inbound ports, two of them are velocity command (i.e., manual, and autonomous) and one is for scan messages (i.e., obstacle avoidance), plus one outbound port exposing the current speed of the robot to the system. While the architecture was designed using the existing autonomous wheelchair as a reference, this subsystem creates an abstraction of the robot platform and can be replaced with a different platform to transfer the wheelchair architecture to other robots.

3.5 Main system

The model of our architecture divides most of the elements in subsystems, while this is useful to create an abstraction in the architecture, it is not strictly required, especially true in small architectures or prototypes. In these cases, a model-based approach is still viable, but it may require a less detailed description. In our architecture there are two elements which are modeled directly in the root system: the *overseer* node, which manages the global state machine and it is not strictly related to any subsystem, and the virtual bus, which represents the ROS communication infrastructure and all the connections modeling topics are bound to this component.

4 THE NODE IMPLEMENTATION

As presented in [2], nodes are described using a *base node* as a starting point. This template model includes some of the fundamental element of a ROS node and provides additional features to make it more robust and reliable. Fig. 2 shows a simplified UML model of the C++ classes composing it.

4.1 Base node structure

The base node root superclass is called *LifeCycle* and it is a general class for process life-cycle management based on a finite-state
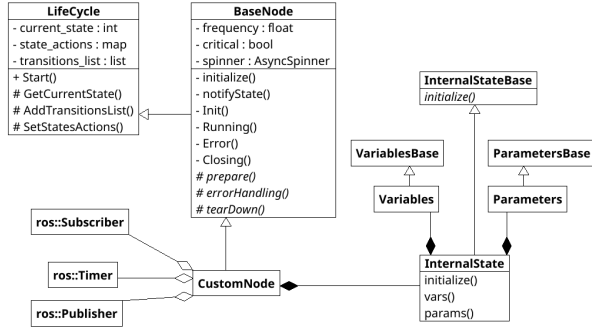
Figure 2: Simplified UML representation of the class structure implementing the nodes created using automatic code generation.

machine. It is implemented using a list of pairs (source state and destination state) to define transitions, no input is expected to trigger the transition. Each state is bound to a function or class method, which is executed after transition into that state; the main execution loop of the node is bound to a state with a self-loop. The design of the *LifeCycle* class is general and not bound to the concept of ROS nodes: states, transitions, and functions are defined by the developer; this is to simplify future variation or extensions of the base node or to use the same class framework to implement non-ROS components in the architecture.

The *ROSNode* class extends *LifeCycle* and implements all the basic elements of a ROS node. Here, the actual life-cycle of the node is defined and its methods are bound to their corresponding state. The evolution of the node is quite straightforward.

- **Init**: this is the initial state of the the node and it comprises two steps. The first step is a common initialization which applies to every node, it initializes ROS and it defines the asynchronous spinner [2]. If this routine is successful, then a specific initialization procedure of the node is executed; this procedure is an abstract method implemented in the child class.

- **Running**: if initializations are successful, the node moves to this state. Since the spinner is asynchronous, this state repeats at low frequency a check for errors or node termination. It is possible to change the main loop frequency using a parameter, but it will only change how fast the node reacts, specifically, how much time will pass from the error generation, or termination, to the actual state transition.

- **Closing**: ROS nodes can be closed as any other process by sending termination signals. Since some nodes may handle hardware connections or other routines requiring a specific shutdown procedure, the base node switches to this state after capturing a termination signal. This method contains all the ROS default shutdown commands right after a call to an abstract method for custom shutdown procedures.

- **Error**: whenever an error is detected, the transition to this state is triggered. In the superclass this method is abstract with a default implementation to manage some common

---

[2] http://wiki.ros.org/roscpp/Overview/Callbacks

initialization error; child classes can implement their own version to manage specific error procedure and redirect the workflow to the preferred state. For instance, in some cases an initialization error has to force a shutdown, but in others it is possible to solve it by reverting to the *init* state.

The base node implements a state notification system using ROS services. In the *init* state a ROS service client is initialized, it calls a service to notify its current state, the call happens at the beginning of each new state. Given the structure of ROS, this system works even if no active service server exists, moreover the use of services simplifies the creation of specialized tools to monitor the life-cycle of the running nodes. It is possible to customize two parameters in the base node regarding state notifications: criticality and frequency. A critical node needs to constantly notify its current state to confirm it is alive and not in an error state; ROS is not an hard real-time framework, but also in soft real-time applications it is important to verify the liveness of the connections, e.g., when sending velocity command to a robot, so the architecture can react accordingly if a node does not respond. The frequency is how often the main loop of the node is repeated; while this value is not connected to any execution loop, since they work independently, it is an important parameter in order to have nodes quickly react to errors and termination.

Any new ROS node can be implemented by extending the *ROSNode* class. A developer, or in our case the automatic code generator, needs to implement the abstract methods: *prepare* for initialization, *tearDown* for shutdown procedure and *errorHandling* to manage errors. Any ROS subcomponent (e.g., subscribers, publishers and timers) has to be declared as attribute of the node, moreover, for each component which has a callback (i.e., subscribers and timers) a corresponding method has to be declared. When implementing a ROS node, the correct place to initialize ROS components, bind callbacks and retrieve ROS parameters is the *prepare* method, since it is executed at the beginning of the life-cycle of the node, but after the ROS initialization; this means that the ROS parameters server is already available, but everything is yet to be executed.

This structure of a ROS node, based on a superclass encapsulating the main ROS functionalities and the separation of framework code from problem specific code, it is also useful when implementing nodes that use communication channel different from ROS. This will be shown in details with an example in Section 5.3, where we will detail how to implement a low-level communication component while being consistent with the callback-based ROS structure.

4.2 Internal state structure

A significant issue when new developers start implementing ROS nodes is managing parameters and variables passing from one callback to another. Parameters are often uncategorized, modified during execution creating inconsistencies, hard-coded as constants in the implementation; at the same time, variables are often declared as global to grant visibility in the callbacks or duplicated in multiple places. Moreover, declaring parameters and variables directly as class members removes the distinction between framework code and problem specific code.

Because of the aforementioned issues, while designing the structure of the base node, we also created a class to encapsulate the

parameters and the problem specific variables of a node. This description of the internal state is based on a super class called *InternalStateBase*. This superclass only contains two members: a structure for variables called *VariableBase* and a constant structure for parameters called *ParametersBase*. Both are defined as shared pointer to minimize memory allocation when passing the internal state to functions, moreover the parameters are constant, this guarantees they will not change after initialization. The *InternalStateBase* class is abstract; it includes a pure virtual method for initialization to guarantee the entire internal state of the node is valid at the end of the *init* state. By extending the *InternalStateBase* superclass it is possible to implement a specific internal state for a node; moreover, by extending one of the two structures, or both of them if necessary, a designer can categorize parameters and variables to be used by problem specific code.

4.3 Life-cycle and global state machine

As said before, the base class for the ROS node implements a strict life cycle with state notification, therefore we implemented an *overseer* node, used to receive this notification, to track the current state of each node, and, in our implementation, to manage a global system-wide state machine. The node itself is designed with flexibility in mind, it is possible to parametrize the definition of the state machine, making it suitable for different architectures, and it also offers two ways to trigger a state transition: shared memory and ROS services.

In our architecture we use the global state machine to differentiate between autonomous and manual driving; this requires the transition to be fast and reliable, something ROS services cannot guarantee, and this is why we implemented also the shared memory mechanism. The choice between the two communication means is completely transparent to the developer; indeed we developed a general interface to encapsulate the communication with the global state machine. During execution it is enough to trigger a state transition (or query the current state), then, depending on the system configuration, one or the other method is used. The choice is based on the deployment of the node and it can vary at start-up; specifically, if a node is executed on the same machine of the *overseer*, then the shared memory approach it is used, otherwise the communication goes through ROS services. The utility of the interface is two-fold, not only it hides the selection of the communication method, but it also separates the interaction with the global state machine form the state machine itself, making our *overseer* node just one of the possible implementations.

5 CODE GENERATION

The model describing our architecture (Fig. 1) includes two different types of nodes: existing ROS nodes and new nodes created specifically for the robot. When considering the code generator, it is necessary to create a further distinction in the second category: nodes that operates using solely ROS for external communication and nodes that requires unique form of communication (e.g., device drivers).

5.1 Existing ROS nodes

One of the most important features of ROS is its community and ecosystem, they provide a vast amount of already implemented nodes covering plenty of functionalities.Given this, it is particularly important not only to be able to include the existing nodes in our model, but also to treat them correctly during the code generation procedure. On the modeling side we solved the problem by describing existing ROS nodes only by their interfaces. This means creating an AADL file representing the target package (i.e., *joy*) and then defining all the nodes as component types, specifying expected subscribed topics as inbound ports, published topics as outbound ports, and services as subprogram calls. In principle, it could be possible to do this process automatically by combining ROS graph and ROS wiki analysis, nevertheless, some human intervention is still required.

When generating the corresponding code, all nodes without an implementation are considered as existing nodes, therefore, their source code is not generated; these nodes are still part of the architecture and therefore they are included in all the launch files depending on the structure of the AADL *system* components. Moreover, the code generator automatically remaps the name of their topics according to those defined in the model. At the current state of development, we still do not manage ROS node parameters for existing nodes directly in the model; we still need to decide if we want to maintain complete support for ROS standard, including YAML files, or implement a conversion from YAML to ASN.1, i.e., the format used for parameters definition.

5.2 Custom nodes

Custom nodes are modeled following the meta-model introduced previously. Each one is a process extending the base ROS node and ROS components are declared as threads. The code generator analyzes the AADL model and, if it contains all the necessary elements, it generates the corresponding C++ ROS code. The code generator is built to identify some specific types of thread, namely: timers, subscribers, publishers (triggered periodically by timers), and subscriber with an integrated publisher; these are already available as templates when including the ROS AADL package in the model to reduce the design time of custom nodes.

Let us take as an example the *follower* node, omitting the actual algorithms, its inner working is quite straightforward: adjust an input velocity command depending on the most recent laser scan and republish it. In the design of the node this translates into a subscriber for the laser and a subscriber for the velocity command, this second component also includes a publisher which republishes the adjusted input. Another example is the *manual_mux*, in this case the node needs to relay one of the two manual input to the output depending on the current global state of the system, moreover it has to output a velocity command at a constant frequency event if the sources are not constantly providing the input. The internal structure, in this case, is composed by two subscribers, one for each input source, and a publisher triggered by a timer running at a specific frequency.

In both cases, the model also has a *data* subcomponent which represents the internal state; here parameters and variables can be included as properties in the model. To maintain the generality
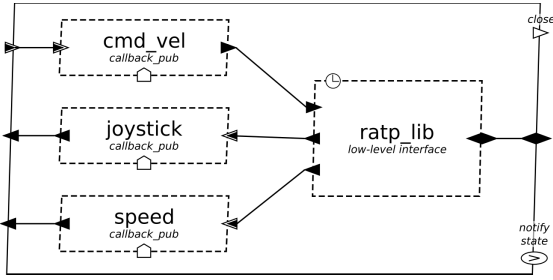
Figure 3: Graphical representation of the AADL model used to describe the *ratp_node*. For clarity, in this representation elements from the base node are omitted (i.e., main thread, state machine, and internal state)

and portability of the model, parameters are described using ASN.1. Other than the name and the type of the parameters, it is possible to specify bounds or a default value. Later, at runtime, the designer can use ASN.1 values to tune parameters specifically for the current execution; these values have to be converted into a ROS compatible format, while this procedure seems overkill, the model is created to abstract from the underlying framework, therefore it is important to maintain a framework-independent data definition. Regarding variables, they are significantly more complex to manage with respect to parameters, since there is no limit to the complexity of the data structures used in a program. Therefore, they are defined directly in the target programming language.

All the features described up to now are compatible with the automatic code generator, it can analyze the model and generate packages, launch files, dependencies, and source code. It also creates, if not already available, all the headers for the problem specific code, that a domain expert can later use as a reference to implement all the necessary algorithms. The code generator works both on AADL and ASN.1, the same toolchain takes the entire model (node descriptions and data descriptions) and creates a working node ready for execution. While the support for various configuration is already enough to cover most ROS nodes, the code generator is an ongoing development and it does not yet generate some ROS features. An example is ROS filters [3] to create synchronized subscribers triggered by multiple topics. Nevertheless, we were able to completely model and automatically generate all the nodes present in our architecture, with the exception of one.

5.3 Special nodes

An exception to automatic code generation is the *ratp_node*; this node is responsible of the communication between ROS and the low-level hardware, which is done using a library provided by the company making the PMK. While the model is powerful enough to describe this type of communication, it is unfeasible to implement the automatic code generation because of a very particular implementation that only works with this specific hardware component. There are two key challenges here: first, be able to capture the structure of the node in the model, second, define the base node in such a way that this type of special situation can be easily implemented.

---

[3]http://wiki.ros.org/filters

For the first case, AADL already offers a solution, because connections, ports and threads can model any kind of communication protocol or execution flow. Fig. 3 shows a graphical representation of the AADL model of the *ratp_node*. In this case, we used an AADL thread component to model the communication thread. This thread has a bidirectional port, representing the communication with the low-level hardware component, which matches the corresponding port of the outer process. As additional input, it receives a velocity command and it outputs the current velocity and the current set-point generated by the on-board joystick, these data flows are modeled using directed ports. These ports are connected using internal connections to the corresponding thread modeling the ROS behavior, however there is an important distinction between these threads and the usual ROS components depicted in the other nodes: as visible from the model, threads behaving as subscribers are triggered by a port on the communication thread instead of a process-level port; in addition, a publisher-like thread output its message directly to the *ratp* thread.

The similarities between ROS subscriber and publisher and the structure of the *ratp_node* does not end with the model. To implement the node we started from the model of a similar version that used only ROS components. Basically all the data flow coming from the low-level hardware were modeled as ROS subscribers, while velocity commands as a ROS publisher, then we used the code generator to create the skeleton of this node. Using the already created structure as a reference, we mimicked the callback structure of ROS to create the bridge between the low-level hardware and ROS messages, by replacing the subscriber bindings with standard bindings triggered by the communication thread. We also re-routed the publisher to create messages compatible with the hardware communication protocol. Moreover, implementing a separate thread to handle hardware communication did not forced us to change the underlying structure of the node since the base node is already implemented using an asynchronous spinner. In conclusion, while this node required a specific implementation by hand, it was easier and more efficient to model a similar node, generate the code and then implement some modification, instead of implementing it from scratch.

6 ARCHITECTURES COMPARISON

With this use case we wanted to prove that is possible to realize a full architecture using our proposed approach for model-based development and code generation. To do so, we took an existing and fully functional robot platform and used our toolchain to replicate the functionalities with an automatically generated architecture based on a model.

Fig. 4 shows the conceptual structure of the original architecture, while Fig. 5 represent the ROS graph derived from the running system; it is immediately visible that they do not match. The reason is simple, as often happens when developing software, features were added directly in the architecture without propagating them in the original project. Later some of these features were removed, but they left some dependencies behind. This problem is exacerbated by the fact that in ROS there is no way to visualize and analyze the structure of the architecture before runtime, all the tools to explore the ROS graph of nodes and topics require the system to be
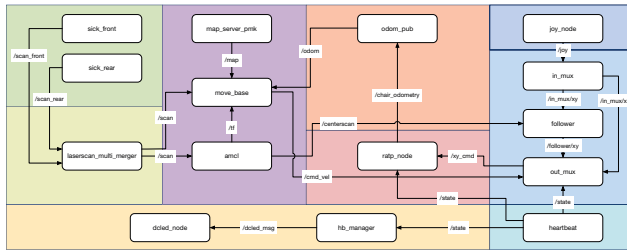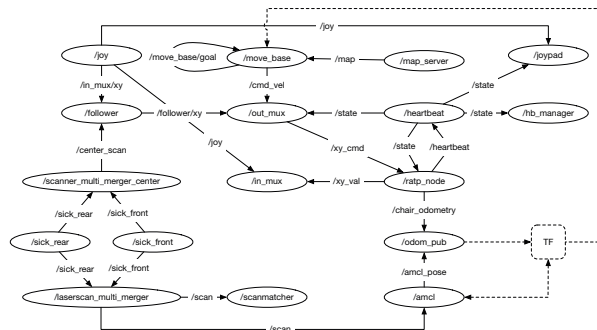
Figure 4: PMK former software architecture.

Figure 5: Nodes and topics connections map in the original architecture.

running. Of course the use of a model does not intrinsically solve the problem, since any developer can always ignore it and add functionalities independently, but, combined with code generation, it creates an environment which encourages good practices and good designs.

Fig. 6 shows the runtime ROS graph of the architecture automatically generated from the model. Other than using this graph to confirm the connection between the model and the actual architecture, it is possible to use it to do a comparison with the former architecture to identify design errors in the original one. Indeed, given the lack of an underlying model, it was impossible to identify most of these errors before runtime, since the ROS graph is not available when the system is not running. This makes these errors quite costly in terms of resources spent to fix them later in the development cycle [8]. In addition, in the generated architecture, topics and connections are defined at modeling stage. This lays the foundation for possible analysis, such as the presence of detached topic, or different message types in communications.

In the hand-written architecture three main issues are visible:

(1) There are some connections that are useless. For instance, the *heartbeat* node communicates its state to an excessive number of other nodes. This translates in a waste of resources, with pointless messages clogging the communication layer. The same applies for the odometry, computed and communicated by more than one node, this could cause inconsistencies.

(2) There is a circular dependency between nodes *odom_pub* and *amcl*. The former publishes the odometry on *tf*, read then by *amcl*, but it also expects an initial pose coming from
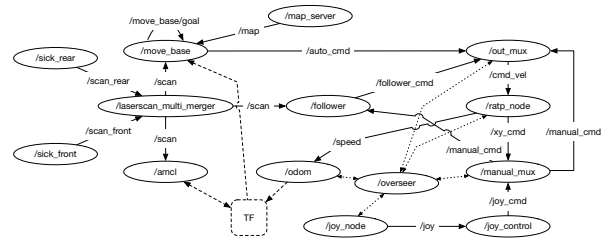

Figure 6: Nodes and topics connections map in the generated architecture.

amcl. This causes an inconsistent initialization that could possibly lead to unexpected results and behaviors.

(3) Nodes *laserscan_multi_merger* and *scanmatcher* are detached from the rest of the architecture. Actually, they are a duplicate of another node, *scanner_multi_merger_center*, meaning they are not used and their running cycles only waste computational power.

Analyzing the ROS graph is useful to provide an indicator of the quality of the software design, but it gives no information regarding the functionalities of the system. Moreover, bad design choices or design flaws not always translate into missing or faulty behaviors. For example, even with the problems of the hand-written architecture highlighted before, the wheelchair was capable of functioning in full autonomous mode with no serious issues, except for minor problems caused by a faulty communication with the low-level control module. To provide an empirical proof of the correctness of the generated architecture, we performed a test in autonomous mode, comparing the wheelchair behavior running with the hand-written architecture, against the generated one. The paths followed are visible in Fig. 7. The two routes are comparable, with a strong similarly also in the wheelchair direction, plotted using oriented arrows. This result is useful to understand that expected behavior, namely the PMK enhancements, can be reached and obtained without writing the ROS nodes code, but only the application specific one and the software architecture AADL model.

7 DISCUSSION AND RELATED WORKS

Developing a toolchain that goes from the definition of a model to a complete implementation is a difficult task. It requires a vertical expertise that goes from deep understanding of model-based design, to detailed knowledge of the target domain, through a solid base of software engineering. That is why our approach does not have the ambition of being the definitive solution to the problem of software development in robotics, but it is an effort to produce something closer to the developers and the problem experts, instead of focusing only on the architectural view. The choice of AADL is also related to this, since the language offers tools to model hardware components and to do low-level system analysis, like latency estimation and resource consumption. This type of analysis is the next step of our work, especially now that we have completed our toolchain and can strongly correlate the model with the implementation. Some work using AADL to estimate the latency in a robot has been already done [3], we aim to expand this by adding resource
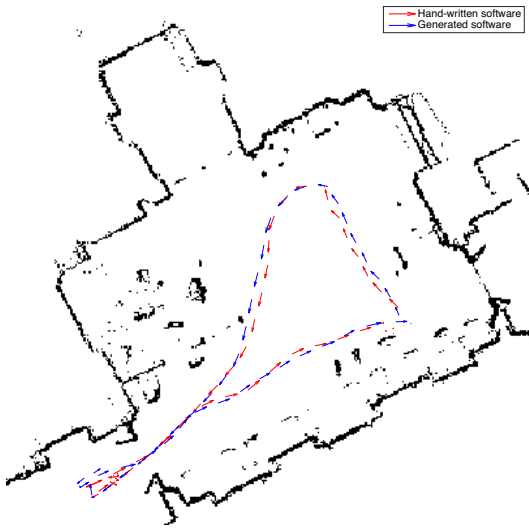
Figure 7: Path followed by the wheelchair in autonomous driving mode. The red arrows show the hand-written architecture route, the blue arrows the generated one. It is visible that they are comparable even though the underlying software is different.

consumption (i.e., bandwidth, CPU) and ROS-specific analysis (i.e., topic compatibility).

Code generation for robotic middlewares it is not a novelty either and various solution exists to automatically generate ROS code. For example, Matlab provides the Robotics System Toolbox that generates C++ code for a standalone ROS node from a Simulink model, but the results are suitable only for prototyping since it is unnecessarily verbose and rely on native Matlab functions. Additionally, some robotics frameworks are built around the concept of model-driven development and automatic code generation. Smart-Soft is a service-oriented component-based approach for robotics software based on communication patterns. The development of components in SmartSoft is based on a DSL, derived from UML, which is used as a starting point to automatically generate various artifacts which lead to the complete components. Other middlewares, like C-Forge [14] and RoCK, use languages with different level of formality to partially generate code. While all these offer a complete or almost complete toolchain from the model to the implementation, they have an obvious downside: the entire process is implemented with the aim of working with a specific middleware. This means they cannot be adapted to a different middleware or used for more general approaches. Some more general approaches exist, an example is BRIDE [7], one of the technological products of the BRICS project, or the work presented in [1], but in both cases most of the effort is at the architectural level, and they lack ways to model the inner working of the components.

8 CONCLUSIONS

In this work we presented a practical use case on how a model-based approach can be used to completely model and automatically generate the code of a robotic platform. The model is written using

AADL, it can describe the hardware devices (i.e., sensor, control interfaces), the software modules and the connections between each component. The target middleware used for code generation is ROS, in particular we provided a basic implementation of a ROS node that includes a well defined life-cycle, a distinction between framework and problem-specific implementation, and a strict description of variables and parameters. With this example architecture we gave an overview of the most common nodes that appears in a ROS-based architecture, namely, already implemented nodes from existing packages, new custom node that rely only on ROS for communication, and nodes that requires a special approach for low-level hardware interactions. With the experimental part, we have shown how the architecture generated automatically not only can replicate exactly the functionality of the one developed by hand, but also highlighted design problems present in the original implementation.

REFERENCES

[1] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2017. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *Robotic Computing (IRC), IEEE International Conference on.* IEEE, 172–179.
[2] Gianluca Bardaro, Andrea Semprebon, and Matteo Matteucci. 2017. AADL for robotics: a general approach for system architecture modeling and code generation. *IRC 2017- IEEE International Conference on Robotic Computing* (2017).
[3] Geoffrey Biggs, Kiyoshi Fujiwara, and Keiju Anada. 2014. Modelling and analysis of a redundant mobile robot architecture using aadl. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots.* Springer, 146–157.
[4] Davide Brugali and Patrizia Scandurra. 2009. Component-based robotic engineering (part i)[tutorial]. *IEEE Robotics & Automation Magazine* 16, 4 (2009), 84–96.
[5] Herman Bruyninckx. 2001. Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, Vol. 3. IEEE, 2523–2528.
[6] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. 2013. 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.* ACM, 1758–1764.
[7] Alexander Bubeck, Florian Weisshardt, and Alexander Verl. 2014. BRIDE-A toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of.* VDE, 1–6.
[8] Maurice Dawson, Darrell Norman Burrell, Emad Rahim, and Stephen Brewster. 2010. Integrating software assurance into the software development life cycle (SDLC). *Journal of Information Systems Technology and Planning* 3, 6 (2010), 49–53.
[9] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. 2012. Robotml: a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots.* Springer, 149–160.
[10] Peter H Feiler, David P Gluch, and John J Hudak. 2006. *The architecture analysis & design language (AADL): An introduction.* Technical Report. DTIC Document.
[11] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. 2008. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 4 (2008), 42.
[12] Sylvain Joyeux. [n. d.]. Rock: the robot construction kit. ([n. d.]).
[13] Gergely Magyar, Peter Sinčák, and Zoltán Krizsán. 2015. Comparison study of robotic middleware for robotic applications. In *Emergent Trends in Robotics and Intelligent Systems.* Springer, 121–128.
[14] Francisco J Ortiz, Diego Alonso, Francisca Rosique, Francisco Sánchez-Ledesma, and Juan A Pastor. 2014. A component-based meta-model and framework in the model driven toolchain C-Forge. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots.* Springer, 340–351.
[15] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.
[16] Christian Schlegel, Andreas Steck, and Alex Lotz. 2011. Model-driven software development in robotics: Communication patterns as key for a robotics component model. *Introduction to Modern Robotics* (2011), 119–150.