GIANLUCA BARDARO

ROBOTS, SOFTWARE AND OTHER STUFF

bip, bop.

— A robot

ACKNOWLEDGMENTS

Thanks all.

ABSTRACT

We did things, we saw people.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

The story so far: In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.

— The Restaurant at the End of the Universe, Douglas Adams

Robotics popularity is growing everywhere. Not only in the academic world, where a lot of research is now applied to robotics, but also in the industrial world, where new companies are providing commercial solutions involving robots. Even the general public is now more used to a society where robot coexist and collaborate with humans. The first model of iRobot Roomba was introduced in 2002, this means that today there are young adults that only know a world where robots in the household are the norm.

The current evolution of robotics as a field is similar, in a way, to the growth in popularity of mobile phones, first, and smartphones, later. Originally, the idea of a personal wireless communication system was only possible in science fiction, then scientific progress and new technologies made it possible. At fist only for very few applications (e. g., the military, the railroad system), but later it grew exponentially and today it is part of our everyday life. Many factors made this leap possible: first of all, technological advancements, like miniaturisation, battery life extension, increase in display quality, cheaper computational power, additionally, a sense of need, people felt that a mobile phone was a great addition to their life, lastly, standardization, shared communication platforms, accessible development environments, and multiple abstraction layers.

The same was for robots, originally no more than toys, mechanical puppets and mysterious automata. They existed, as truly autonomous agents, only in the minds and works of writers and directors, and even today we are not able to match those visions.

As soon as technology made it possible, the first autonomous arms were developed. Initially applied to heavy industry to replace human in dangerous and highly specialized tasks, later, technical refinements and functionality extensions made them suitable for healthcare and the military. From here it was an explosion of different technologies, shapes and applications. Autonomous arms evolved in precision, power and dexterity, from the massive industrial arms, to the agile surgical robots. Soon after the development of the first complex arms, many researchers tried to realize the vision of a full humanoid robot, but, even today, after many progress we are not able to fully replicate the complexity of the human body. Mobile platforms were the next logical step, robots able to autonomously explore and navigate the environment, robots able to reason on what they detect and to react accordingly.

In the last two decades, robotics have been applied in numerous fields and robots assumed a myriad of shapes and functionalities. In industry, robots are used for welding, painting, drilling, cutting, handling dangerous materials, moving heavy objects, pick-and-place, inventory management. In healthcare, today, surgical robots are the norm, but advancement in soft robotics made robots suitable for rehabilitation and elderly care. Most of the recent discoveries of planetary science exist thanks to rovers, autonomous mobile robots that can, unassisted, explore the surface of planets, asteroids and comets. Moreover, maintenance in outer space is extremely dangerous for humans and often impossible, only robotic arms and autonomous probes can perform them. Back on Earth, in our houses and cities, robots are not an unusual sight. There are robotic vacuum cleaners and lawn mowers, autonomous robots deliver packages directly to the front door and self-driving public transportation is a reality in various cities. Fully autonomous cars are still only prototypes, however not because of technological limitations, but mostly for economical, social and legal reasons. Thanks to the recent progresses in human-robot interaction, the sight of a robotic waiter or concierge, while marvellous, is not completely unexpected. Lastly, unmanned autonomous vehicles (e. g., off-road vehicles, drones, boats, submarines), have been used successfully in search and rescue missions and to operate in dan-

gerous environments or unreachable by humans, like mountain peaks, volcanos, disaster zones, and contaminated areas.

In this brief history of robots, most of the progress and technological advancements seems related to hardware. More responsive motors, more precise and reliable sensors, cheaper electronic and computational power. All these advancements contributed to what is robotics today. However, software has always been one of the main concerns of any roboticist. The implementation, the logic, is what makes the difference between a mechanism and an intelligent robot. Since their inception, robots have spawned a series of software solutions to implement their functionalities. For example, the Stanford Research Institute Problem Solver, better known by it acronym STRIPS, is an automated planner developed for Shakey that became the foundation of modern action languages. Modern robots have software architecture far more complex than Shakey, they coordinate multiple sensors and actuators, moreover they implement different functionalities and are expected to operate in real time.

This is why, more recently, more or less in the last twenty-five years, a lot of efforts in robotic software revolved around the design of a solution to streamline and simplify the development process. The answer was the introduction of robotic middlewares and frameworks and to rely on component-based designs. This approach fits perfectly the necessities of robotics, components encapsulate functionalities and promote reusability, while a predefined communication layer frees the developer from the burden of micromanaging the low-level interactions. After the first wave of ad hoc implementations, few frameworks rose in popularity and become standard de-facto for robotic software development. Today, depending on the specific application, a developer can choose various framework or middleware: OROCOS (or its derivation RoCK), for hard real-time application, SmartMDSD, for a more complete and structured development environment, YARP, for a more light-weight and data-centric approach, or ROS, for more extensive support and development freedom.

Middlewares and frameworks have fuelled the progress of robotic systems, creating the current scenario of robot design and development. Hundreds of components are already available to

anyone who wants to implement his own robot and experts can setup the most common functionalities (i. e., teleoperation, mapping, indoor localization and navigation) of a new system in a matter of days. However, the learning curve to reach this kind of expertise is quite steep and extending the functionalities of a robot beyond what is currently available requires a considerable effort not strictly related to the new functionality itself. By doing again the parallel between robotics and smartphones, we are currently in robotics in the same situation developers were before the standardization introduced by Android. Today, an Android developer can bootstrap and deploy a new application on millions of devices in few steps, thanks to abstraction layers that separate the development environment to the underlying hardware and operating system and thanks to advanced design, development and simulation tools. Of course smartphones are not robots, while there is a great variability from one device to another (e. g., screen size, quality and number of cameras, sensors availability, type of mobile network, etc.), they cannot be compared to the incredible range of sensors, actuators, shapes and functionalities that exist in robotics. For this reason, while robotics can aim to achieve the same streamlined development of smartphones, the approach needs to be different.

1.1 MOTIVATIONS

Middlewares and frameworks created the present development environment of robotics, but current approaches are not suitable any more for a constantly advancing robotic field. The personal experience of an all-around robotic expert still drives robotic software design and development. When developing a new system or application it is expect that a developer has total expertise on the low-level functionalities provided by the underlying framework and the high-level functionalities to be implemented; while this was possible in the past, it is an unsustainable approach today. Not only it is necessary to create a distinction between different roles in the design and development process of a robot, but is is

also necessary to provide to these roles the right tools to fulfil their tasks.

The *system designer* needs tools to outline the architecture of the system and describe the high-level interactions and requirements of components. This can be achieved using a modelling language to describe components and their inner workings in an agnostic way with respect to the underlying framework. This approach, not only provides the right environment for the designer, but it also provides early detection of errors, an architectural overview of the system and system-level reusability.

The *component developer* should focus only on the implementation of the internal logic and not on the structure of the component itself, since this is the role of the designer. To do so, the component developer needs an environment that abstracts from the framework-related boilerplate code and provides a contained development space. Potentially, the logic implemented should be portable from one component to another, even if they are not based on the same framework, given they share the same design principles. Building on top of the modelling language used by the system designer, it is possible to achieve the ideal development environment by delegating to an automatic code generator most of the boilerplate implementation, and by defining a bounded reference component that can be used by the component developer as a starting point.

The *application developer* implements high-level functionalities, that should be independent from the underlying architecture of the robot. In practice, this means it should exist an abstraction layer between the low-level capabilities provided by one or more components and the high-level applications. There is a plethora of robots, with different configurations and implementations, however it is possible to abstract most of the capabilities independently from the system. An example could be teleoperation: by defining linear and angular velocity of the mobile platform it is possible to control any robot, independently from their physical configuration. Using these general interfaces the application developer should be able to implement high-level functionalities for multiple robots with minimal modifications. In order to achieve this it is necessary to define the concept of capabilities, to identify them

in a robot architecture and to provide a framework-independent way to interact with them.

1.2 THESIS CONTRIBUTIONS

Our proposed approach revolves around two key factors: formalise the design and development of robotic software and streamline the implementation process for the different experts involved. To do so, we developed a collection of standards, tools and techniques, each one focused on a different aspect or phase of the design and development process to assist each role on their specific task, but all interconnected together to benefit one from another.

For the *system designer* we exploited an existing modelling language to create a suitable description for robotic architectures. We relayed on the fact that the most popular middlewares and frameworks adopted a *component-and-connector structure* to create a generalized approach. Since the aim is to cover the entire development process by supporting all the actor involved, we then focused on creating a more specialized description to model ROS-based architectures. The generalized approach already covered the concept of components (i. e., nodes), ports (i. e., publishers, subscribers, service clients and servers) and connections (i. e., topics and services), while the specialized description goes more in details by providing model for messages and the internal functions of nodes. Moreover, we tried to capture some relevant robotic design pattern, both outside the component (e. g., topic multiplexer) and inside (e. g., message relay). The advantages for the system designer are multiple; a model of the complete system gives an architectural overview which is otherwise impossible to achieve before runtime, moreover it is possible to check, before execution, the compatibility of the communication channels, a functionality that is usually unavailable in those frameworks and middlewares that connect the component at start-up time. Additionally, the designer can rely on a library of already existing templates, this makes the design of the system easier and the

resulting architecture more robust. Lastly, by basing this work on an existing modelling language we give the designer the opportunity to exploit all the other tools available for the language, few examples are: latency estimation, computational load, hardware allocation and fault propagation

The *component developer* often works together with the *domain expert*. With our work we provide support for both roles and their interaction. From the model created by the *system designer* we provide an automatic code generation to ROS. The target implementation is based on a reference node specifically engineered to minimize the amount of boilerplate code, moreover, it provides additional features that are usually borne by the *component developer*, few examples are: internal life cycle of the node, well defined initialization procedure, encapsulation of parameters and internal state, clear separation between the middleware and implementation. The latter is particularly important for the role of the *domain expert*; their contribution to the functionalities of a robot is fundamental, they provide control software, local and global planning algorithms, robot behaviour, and more. Since they are expert of a specific domain and carrier of specialized and valuable knowledge, they often do not and ideally should not implement the component directly, but to have access to a suitable interface. In our proposed model and automatic code generation approach a *domain expert* can implement the functionality independently from the component and then embed it in the model, the automatic code generation will include it in the final implementation.

Lastly, for the *application developer* we developed the concept of robot capabilities. We define them as low or medium level functionalities (e. g., directional movement or navigation) and a developer can use them to interact with the robot (i. e., to send commands of varying complexity) and to receive information from the robot (i. e., to read sensor measurements). The capabilities are defined manually by analysing the configuration and functionalities of different types of robots (i. e., mobile platforms, drones and mobile manipulators), but the active capabilities on a running system are extracted automatically by analysing the ROS graph. On top of the concept of capabilities we developed an abstraction layer to decouple the application from the underlying

middleware or framework. In our approach we implemented a bridge between the capabilities and, consistently, ROS based system. To do so, we developed a dynamic node that can manage a bidirectional communication with an external system through different communication channels. We provide dynamically defined Python based API where a developer can interact with the robot through capabilities, moreover, we created a set of remote API where JSON messages can be used to trigger capabilities remotely. To test the effectiveness of this approach in simplifying robot development we created a web interface that can be used to create visual algorithms to program a remote robot.

Even if it is not evident at first glance, all these approaches, techniques, standards and tools are all part of a continuous design and development process. The *system designer* uses the modelling tools and templates to define the architecture of the system. He can embed directly the reference to the source code developed by the *domain expert*, and, using properties, even enrich the component with their evoked capability. Through automatic code generation most of the source code is already available with minimal effort, at this point the *component developer* can finalise the implementation by adding anything that cannot be automatically generated, for example special interfaces with the hardware components or specific initialization and shutdown procedures. The result is a robust system where all the components are known, well designed and well implemented, this is the suitable starting point for an *application developer* to exploit safely the abstraction layer defined using robot capabilities.

1.3 THESIS OUTLINE

This thesis is divided in eight chapters:

- Chapter 2 gives an overview of technologies, techniques, methodologies and approaches related to the work presented in this thesis. We focus mostly on software engineering and model-driven approaches, and how they are used in general and robotics applications.

- Chapter 3 provides details of the two main technologies used in this thesis: the Robot Operating System (ROS) and the Architecture Analysis & Design Language (AADL).

- Chapter 4 presents our model-based approach. First we introduce an high-level description of a generic component-based robotic architecture, then we show how AADL can be used to model it. The chapter continues by extending the approach to ROS; initially by describing a single node, an then by designing a complete architecture. We conclude by finalising the model definition using two different and interchangeable data modelling languages.

- Chapter 5 presents our code generation toolchain. First we describe the target of the code generation process, in particular how we engineered an enhanced ROS node with additional functionalities, and how it can be used to implement custom ROS nodes. All the phases of the two-step code generation process are described next. The chapter closes with a complete example from the AADL model to the ROS/C++ implementation.

- Chapter 6 presents our ontology-based robot abstraction. The chapter opens by describing the ontology used to define ROS, the binding between ROS elements and the capabilities, and the capability taxonomy. The abstraction is used to define robot APIs that can be used to exploit specific capabilities. We conclude by analysing how the ontology-based abstraction benefits the model-based approach and vice versa.

- Chapter 7 focuses on the experimental evaluation of our work. We present two scenarios. One uses the architecture of an autonomous wheelchair to demonstrate the functionalities of our model-based development approach. The other exploit our ontology-based abstraction to create a set of web APIs that developer with no experience in robotics can use to program different robotic platforms.

- Chapter 8 draws relevant conclusions of this work and present potential extensions and future works.

1.4 PUBLICATIONS

- **G. Bardaro**, M. Matteucci
  Using AADL to model and develop ROS-based robotic application [7]
  International Conference on Robotic Computing (IRC), 2017

- **G. Bardaro**, A. Semprebon, M. Matteucci
  AADL for robotics: a general approach for system architecture modeling and code generation [9]
  Journal of Software Engineering for Robotics (JOSER), 2017

- I. Tiddi, E. Bastianelli, **G. Bardaro**, M. d'Aquin, E. Motta
  An ontology-based approach to improve the accessibility of ROS-based robotic systems [89]
  Knowledge Capture Conference (KCap), 2018

- I. Tiddi, E. Bastianelli, **G. Bardaro**, E Motta
  A User-friendly Interface to Control ROS Robotic Platforms [88]
  International Semantic Web Conference (ISWC), 2018

- **G. Bardaro**, A. Semprebon, M. Matteucci
  A use case in model-based robot development using AADL and ROS [10]
  International Workshop on Robotics Software Engineering (RoSE), 2018

- **G. Bardaro**, A. Semprebon, A. Chiatti, M. Matteucci
  From Models to Software Through Automatic Transformations: An AADL to ROS End-to-End Toolchain [8]
  International Conference on Robotic Computing (IRC), 2019

2 RELATED WORKS

2.1 SOFTWARE ENGINEERING

The definition given by the IEEE Computer Society for software engineering is: *"the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software"* [2]. It is quite a broad statement, in fact, during the years, various experts gave their own definition [27, 75, 85] of what makes software engineering different from software development. Independently from the specific definition used to describe the field, it is undeniable that software engineering was born and it is used today to design, implement and manage complex computer programs.

From its humble inception in the 1960s as a label to legitimate the work done by the programmers in the Apollo missions, today software engineering evolved in a very complex field, including a long list of subdisciplines. Specialised software engineers work through the entire development process, starting from defining the software requirements, then creating the design, supervising the implementation, designing the testing approach, providing maintenance, and more.

Currently, robotics may not be ready to fully embrace the methodologies and techniques provided by software engineering, and at the same time, software engineers may not have all the necessary answer for roboticists' problems. However, given the rising importance of software in robotics [28], the overlap of the two world is inevitable. Indeed, robotics already benefited of this intersection by adopting component-based architectures [76, 80], software product lines [49], model-based designs [33], and automatic programming [61]. Unfortunately, often these approaches fail to prosper in the robotic community and remain relegated to the academic world.

In this section, we analyse the software engineering side of these approaches, to understand they foundations, which affinity they had with robotics that pushed designers and developers to adopt them and how they can be integrated even further.

COMPONENT-BASED SOFTWARE ENGINEERING

Component-based software engineering (CBSE), also known as component-based development (CBD), is a software development approach where there is a strong emphasis on the separation of concerns. The architecture is divided in components, they encapsulate specific functionalities and interact between each other using pre-defined and clear interfaces.

The idea that software should be componentised is as old as software engineering itself, since it was originally proposed in the first NATO conference of software engineering in 1968. However modern component designs build on prior theories of architectures, frameworks and design patterns. In particular, they can be seen as the natural evolution of object oriented programming. The two development paradigm share various characteristics, for instance functionality encapsulation, information hiding, and well defined interfaces. In fact, often a component is implemented by a single object, or a main object defining the structure and multiple secondary objects to implements its functionalities.

A software component is very versatile, since they can be any kind of software entity, from high-level (e. g., software package or web service) to finer granularity (e. g., software module or web resource). However, the component needs to be consistent inside the same architecture, since they all follow the same component model. The component model is a specification of the concept of component to adapt to a given implementation. In most cases the definition of the model is quite loose (e. g., limited to the interface) and not formally specified, often imposed through other means (e. g., inheritance, forced parametrisation). However, when the component model is strictly formalised, CBSE is a stepping stone for model-based software engineering, a starting point to introduce the concept of formal methods to developers.

The advantage in the use of a component-based approach is an architecture that is extremely modular, functionalities are encapsulated, flexible, components communicates through interfaces, robust, unit testing and limited error propagation, and reusable, components are easy to port and replace. Robustness and re-usability are the two main characteristics of component-based architectures. Both are boosted by the encapsulation and the use of interfaces. Components are tested separately, and the behaviour of the architecture is simulated using the well-defined interfaces. Limiting the amount of functionalities in each component reduce the risk of faulty behaviours, streamline the development, and provides a clear overview of the role of the component in the architecture.

Robotics took great advantage from the characteristics of CBSE, the most popular approaches for robot software development use components as their foundations [19, 20]. This is because of the intrinsic modular nature of robots, which is perfectly aligned with CBSE philosophy. Moreover, in robotics is quite challenging to perform testing, given the interaction with the real word, hence, an architecture where a sub-functionality can be replaced without modifying the supporting infrastructure is extremely beneficial. Unfortunately, the current state of the art in component-based robot software development is polarised on one of the two aspects of components. On one side, developers exploit the re-usability without pairing it with the expected robustness. For instance, this is happening in ROS, since it provides a vast library of existing components, but with total freedom left to the developer. On the opposite extreme, component models are significantly bound to the target architecture and framework that the re-usability is limited to that specific scenario. This is the case of SmartMDSD, they push for a more model-driven design, but the result is a constrained development environment.

MODEL-DRIVEN SOFTWARE ENGINEERING

Model-driven software engineering takes the paradigm that a program is made by algorithms and data structures [94], and evolves it in the concept that a software is made by models and trans-

formations [16]. Models are used as a foundation for the design of the software and to support all the phases of the development cycle. They abstract specific aspects of the software to make it more understandable and manageable. Transformations are a set of rules defined by the software engineer that use the models as input to create programming artefacts or other, more detailed and more focused, models, in a recursive process that lead to a complete architecture. In summary, the software engineer goes from being a program developer to a system designer.

While this methodology seems more complicated than a traditional development approach, given it requires to learn modelling on top of programming and to introduce multiple new steps in the development process, it is actually extremely useful to manage complexity [82] and it increases efficiency and effectiveness in software development [3].

The most known and used application of MDSE is software development automation [83], where model-driven technique are employed to automate as much as possible of the software development process, from requirement definition to the system deployment. The process starts by a single or a collection of models and using a sequence of model-to-model transformation, where each phase use as an input the model (semi)automatically generated in the previous one, eventually manually refined by the specific domain expert. The last step is a model-to-text transformation where the chain of refined models converge in the final implementation. To achieve this result it is necessary for the model to be executable [66], which means have all the necessary information to define an executable program.

This use case is also the focus of our work, since our main goal is to enchanter and streamline robot software development by hiding to the developer the unnecessary complexity introduced by robotic frameworks and middleware. However, this is only the one aspect of MDSE, since models can be used as a *lingua franca*, not only between different domain expert working together on the same architecture, extremely important in robotics, which is often an overlap of multiple fields, but also between different technologies. For instance, ROS promotes thin implementations to increase its interoperability with other frameworks, however,

when this happens it is always through manually implemented bridges that requires a lot of ad hoc conversions.

Model-driven approaches have been successfully used to promote system interoperability [26], by abstracting relevant interfaces from the involved systems and providing a common communication ground. We use this strategy in this work with the support of ontologies (strictly related to modelling techniques [53]), to create an abstraction layer on top of an existing robot architecture to create a generalised interface.

Another important use of MDSE is reverse engineering [21, 77]. Since models can be used to abstract software and highlight important aspect, they are the perfect tool to analyse legacy system and identify significant features, interfaces and functionalities that can be ported in a new system. While in this work we do not directly face this problem, our approach takes strongly into account the already existing solution and software artefacts for robotics. In particular, we model all the existing ROS components and messages, in a platform independent language, this representation could be used in the future to support the migration of ROS implementations to ROS2.

SOFTWARE PRODUCT LINES

Software product lines are software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [70]. They are extensively used in automotive software development, and are adopted in multiple domains.

Conceptually, they are derived from the product line approach used in manufacturing, where product with similar characteristics are bundled together in a family of designs. These design families are manufactured in the same factory, where reusable parts are combined together to obtain one of the possible configurations of the product line.

In the same way, software product lines are a collection of similar software products that share a list of configurable features. A feature can be anything that specialises the software: a module,

an algorithm, a service, a component. The product family share a reference architecture [68] that is used as the backbone of the configuration and specialisation process. All the possible product configuration and the configurable features are modelled using feature trees [36, 67]. The features codified by the tree are usually targeted to functionalities (e. g., different type of communication, different planning methods, etc.), however, non-functional, yet measurable, properties have been used in some works [12] to specialise a product family.

Usually, a complete configuration of a product is a path from the root to one of the leaf of the tree, however, more complex configurations exists where two features can be implemented in parallel in the same product. In this cases, the final design of the software is determined by an overlap of all the complete root-to-leaf paths.

Following the same philosophy of the more constrained manufacturing product lines, the configuration of products in software product lines is defined at design-time. This means that after deployment, the software features are final and cannot be changed. However, software development is more flexible than traditional manufacturing, therefore exist an extension of SPLs that implements a dynamic approach to feature definition: dynamic software product lines. In DSPLs the definition of the architecture is postponed until runtime. Additionally, depending on the target platform, it is possible to "hot swap" the features during execution, adapting to the necessity of the software. SPLs is more in line with a monolithic development approach, where the entire software is completely defined before deployment, while DSPLs are more suitable for flexible approaches such as CBSE.

In fact, component-based software engineering is not in contrast with software product lines, on the contrary, the two approaches complete each other. CBSE promotes software reuse, however, it is often an opportunistic approach, components are developed for a functionality in a specific architecture and then shared with an extended community in the hypothesis of reuse. While this opportunistic reuse is successful in many cases [57], combining CBSE with SPLs lead to a more focused and forward-looking development. At the same time, a component-based architecture is ideal

for implementing the philosophy proposed by SPLs, in particular considering the dynamic version, since plenty of component-based approaches support dynamic connection between components.

While in this work we will not discuss SPLs directly, many of our approaches can supports and are compatible with the feature system of software product line. For example, the bounded design of components, the templatisation of designs and the classification of functionalities. Moreover, SPLs have been successfully used in robotics for architecture modelling [18, 49] and for modelling of manipulation and grasping [11].

2.2 GENERAL-PURPOSE MODELLING LANGUAGES

Models are an abstract representation of a real system or a phenomenon [79]. They are used in all fields of engineering, and, with no surprise, are central to model-driven software engineering. Models can appear in two forms: graphical representations (e. g., the blueprint of a building), or textual descriptions (e. g., a communication protocol description).

Model, whether they are graphical or textual, are defined using a modelling language, which is an artificial language that can be used to express information and knowledge about a phenomena or a system using a consistent set of rules. A plethora of different modelling languages for software development exists, they cover different aspects of the architecture, or different phases of the development, or a targeted to specific platforms or subsystem. Often modelling languages try to provide a "one-size-fits-all" approach by achieving different levels of generality.

UNIFIED MODELING LANGUAGE – Probably the most known, and surely the most popular [65], general-purpose modelling language is the Unified Modeling Language [78]. UML is defined, developed and maintained by the Object Management Group (OMG) [71], a not-for-profit technology standards consortium founded in 1989. UML has been evolving since the 1990s and has its roots in the

object-oriented programming paradigm, nowadays, the current version of UML can be used to represent various programming and system artefacts. This makes UML a true general-purpose language, or almost a meta-language, since it can be specialised and refined for a specific domain using profiles [45].

The profile system in UML provides a generic extension mechanism to customize the language for specific domain or platforms. It allows to refine the standard semantic in a strictly additive way, preventing the extension from contradicting the general semantics. Given the extreme generality of UML and the size of the language, profiles have been used to design more focused modelling languages targeted to specific domains.

The Systems Modeling Language (SysML) [44] is a profile aimed to system engineers that supports design, analysis and validation of system architectures and system-of-systems designs. The design of SysML is based only on a subset of UML, and it tries to move away from its software-centric approach in favour of a more system-centric design. Another example of domain specific profile is Modeling and Analysis of Real Time Embedded system (MARTE) [39], where UML is specialised to model non functional properties, physical constraints (e. g., time, mass, energy) or memory management in real-time and embedded systems. The list of available profiles is quite long, with applications in web development (WebML [25]), service-oriented architectures (SoaML [35]), software product lines [95], data warehouses [64] and many more [4, 42]. Unsurprisingly, the long list of UML profiles includes on specifically designed for robotics: the Robot Modeling Language (RobotML) [32].

Given the scope of this work, we considered RobotML as a possible target modelling language for our approach. However, since it is based on UML, it is strongly software-centric, disregarding the importance of hardware descriptions in robotics. Moreover, the profile is only partially defined and lacks the sufficient documentation to use it as reliable and robust modelling language.

In summary, given its generality, UML and its profiles have been used as a silver bullet to solve any kind of modelling problem. However, the use of general purpose modelling languages, especially in extreme cases such as UML, may hinder the design

approach instead of supporting it [15, 84]. Mostly because of the additional time necessary to learn the modelling languages on top of the domain-specific knowledge and because of the extreme abstraction between the model and the real application. Moreover, in UML, the profiles are powerful tools, but as often happen when significant customisation is available, they inevitably expand causing cross-dependencies [37], nested specifications [39] and conflicts.

SIMULINK – Another well known general-purpose modelling environment is Simulink [29] developed by MathWorks and based on MATLAB. Simulink is far beyond the definition of modelling language since it implements a fully fledged modelling, simulation, and code generation system deeply integrated with MATLAB. It can be used to model an extreme variety of systems, from electronic circuits to image processing pipelines, passing through signal processing and analogue or digital control systems. It is based on a graphical block diagramming tool, originally designed to design control systems, and now extended to support a variety of different applications. Simulink embodies "one-size-fits-all" approach with its generalised modelling interface, this makes the interaction between models of similar domains simpler, but forces the designer to be extremely specialised and proficient in its modelling approach.

The strength of Simulink is also its significant downside. The strong integration with MATLAB makes the modelling suite extremely powerful, but also extremely limited to the functionality implemented by MathWorks. The freedom of choice of the developer is minimal, this means that all the phases of the modelling process have to happen in the boundaries defined by MATLAB. While this approach can be suitable for industrial setting, specific domains or high-level simulation, it is incompatible with the fast changing scenario [28] of modern robotic applications.

ARCHITECTURE ANALYSIS & DESIGN LANGUAGE – The Architecture Analysis & Design Language (AADL) [40] is a modelling language for architectures of safety-critical, embedded and real-time systems. Differently from other modelling languages, which focus on a graphical or textual description, AADL supports both, they

can be used depending on the development phase or the required precision of the design. Due to its emphasis on the embedded domain, AADL contains constructs for modelling both software and hardware components. This architecture model can then be used either as a design documentation, for analyses [41] (such as schedulability and flow control) or for code generation [55] (of the software portion). While it is targeted to embedded system, AADL can be considered a general-purpose language, since its semantic is general enough to be used for various applications, from telecommunication [31] to space [74].

AADL have been successfully used in robotics to perform analysis on latency and error propagation, both before the implementation of the architecture and after. In the former [14], an autonomous wheelchair has been modelled to define the redundancy of the system based on the expected error propagation. In the latter [63], an existing robot implementing a ROS-based architecture has been modelled to estimate the latency of the system and configure the nodes accordingly. However, outside of the approach presented in this work, AADL has yet to be used as a modelling language to describe robotic architectures and as a support tool for the entire development process. This is probably related to the fact that AADL is considered a niche modelling language with limited applications (e. g., latency analysis), in contrast to approaches such as UML.

2.3 DOMAIN-SPECIFIC APPROACHES

Domain-specific approaches vary in complexity and connection to the domain depending on how much they are defined specifically for a given domain. The most general form of domain specialisation is the concept of a Domain-specific language (DSL) [43]. A DSL is created by defining a programming language with a limited expressiveness designed specifically for a particular set of tasks. Often, the level of specialisation is such that the language is tailored on a specific category of users [93]. DSL are often derived from a more general language used as a guideline, another typ-

ical example is the markup meta-language XML [54] that can be used to define specific languages following a predefined schema. However, when used for larger domains they are usually created from scratch, to the point that it is difficult to recognise them as domain-specific (e. g., HTML [51] or AWK [34]).

While special-purpose computer languages have always existed, the term "domain-specific language" has become more popular due to the rise of domain-specific modelling. In fact, modelling benefits significantly from DSL, since they can be used to reduce complexity and to provide a dialect of an original general-purpose language to target a specific domain, a clear example is the use of profiles in UML. Given the importance of DSL in modelling, there are various MDSE environments enabling their creation. For instance, the Eclipse Foundation provide a collection of tools under the Eclipse Modeling Project [52] where Ecore [81, 86] is used for meta-modelling, Papyrus [62] for UML profiles and general modelling, Xtext [38] specifically for DSL engineering, and Xtend [13] for code-generation. Other examples are the Jetbrains Meta Programming System (MPS) [58], specifically designed as a complete environment for the creation of DSL, and MontiCore Language Workbench [60], a modular workbench for the design and realization of textual DSLs, which has been used with some degrees of success to create robotic specific languages [87].

In this work we recognise the usefulness of DSLs in model-driven approaches, in fact, we defined ourself one based on XML to create an intermediate representation from AADL (Section 5.4.1). Additionally, while defining our meta-model, we pondered the possibility of define a DSL to reduce the complexity of the model. In the end, we realised that the AADL already provided, through inheritance, tools to manage the complexity of the language, without the need of hindering the freedom of the designer.

AUTOMOTIVE

The Automotive Open System Architecture (AUTOSAR) [46] is the perfect example of a complete domain-specific environment, where the entire development stack (i. e., design, implementation and ecosystem) has been design for a specific field. AUTOSAR is

a worldwide development partnership of automotive interested parties founded in 2003. It pursues the objective of creating and establishing an open and standardized software architecture for automotive electronic control units (ECUs). Goals include the scalability to different vehicle and platform variants, transferability of software, the consideration of availability and safety requirements, a collaboration between various partners, sustainable utilization of natural resources, and maintainability throughout the whole product life cycle [1].

AUTOSAR is based on a three-layered architecture: the basic software, a series of standardised modules necessary to run the functional part of the upper software layers, the runtime environment, a middleware which abstract the network topology for all the necessary communications, and the application layer, the application software components implementing specific functionalities. This structure is not particularly different from what is currently happening in robotics, however, there are two significant difference that make AUTOSAR a industry-accepted standard, while in robotics there is no consensus yet.

First and foremost, AUTOSAR is an industry-created and industry-imposed standard. It is a top down approach where the biggest actor of the automotive industry lobbied together to create a standard. In robotics, the approaches are bottom up self-emerging. Second is the superstructure of tools, methodologies, standards, and practices associated with AUTOSAR [5, 6, 50]. Robotics should learn from the results obtained by the automotive industry, but not by copying the top down approach of imposing a standard, but by providing the necessary superstructure to the existing self-emerging approaches.

SPACE

The development of space application is significantly different from traditional development. Considering the limitation imposed by the harsh environments, the physical constraint impacting the software (e. g., weight of the hardware components), the one-time deployment, and the extreme criticality of the implementations, it

comes with no surprise that approaches, tooling and methodologies for space applications are extremely domain specific.

TASTE is a toolchain targeting heterogeneous, embedded system using a model-driven development approach [73]. In space applications software is usually categorised depending on the final deployment, prototypes are often developed with less restrictions and then deployed using space-grade technologies before a mission. In this scenario, TASTE is a laboratory platform designed for experimenting new software technologies, based on free and open source solutions. TASTE has at its core the philosophy of "not reinventing the wheel", therefore, while it is designed for the very restrictive space domain, all its functionalities are build on top of existing technologies and reusable designs and approaches.

At its core, TASTE is based on AADL for the architectural models and on ASN.1 for data models [74]. Additionally, thanks to its modular design and the property system of AADL, it supports VHDL [69] for hardware design, SDL for behaviour modelling, and Simulink for dynamic simulations. TASTE is a complete toolchain from the model definition, using a superstructure built on top of AADL, to the execution platform, it supports code generation and deployment on real-time platform such as Xenomai [48] and PolyORB-HI [92].

The approach chosen by TASTE is remarkable and extremely interesting. TASTE is developed and maintained directly by the European Space Agency, an organisation that has the power to impose standards to their contractors. However, they decided to adopt existing approaches and use them to support their work. This strategy is compatible with the self-emerging robotic environment, instead of imposing new standards, exploit existing technologies to support popular approaches.

2.4 ONTOLOGIES

2.5 ROBOT SOFTWARE DEVELOPMENT

Robot software development is very different with respect to other domains. To be more precise, robotics does not face unique challenges, but it combines in a single domain multiple issues seen and manage independently by other fields. Robot software is bound to the hardware as much as embedded system, and it may requires the low-latency provided by real-time architecture. Additionally, a wide variety of robotic systems have to perform direct interaction with humans, facing the same challenges of human-computer interactions and user interface development. Robots are equipped with multiple sensors and actuators implementing multiple functionalities, this requires architectures to be flexible and modular. Lastly, robots the interaction with the environment is tightly connected with the internal execution loop of the robot, making the intrinsic uncertainty of the real world a guiding element of robot design. This characteristic is almost unique to robotics, a similar level of uncertainty is found in web applications given the interaction with the Internet. The main difference between the two is that a robot needs to understand and interface with the uncertainty of the real world, while a web application tries to bound and constraint it.

MIDDLEWARE AND FRAMEWORKS

Both the state of the art and the history of robotic software revolves around the concept of middleware or frameworks. Systems used to hide the complexity of the underlying hardware platform and to help the developers in creating, deploying and reusing modular implementations. Right now the landscape for robot development is quite consolidated with the current technologies celebrating their 10^{th} anniversary and the pioneers getting closer to their 20^{th} birthday.

Player [47] is one of the first framework for robotics, its aim was to provide support for the development of device drivers. It

become very successful, especially because it allowed to integrate and reuse third party software [91]. Another key element of its popularity was its twin platform: the Stage simulator. A contemporary of the Player/Stage project is the Open Robot Control Software Real Time Toolkit (OROCOS-RTT) [22], a framework designed to provide real-time capabilities for industrial robots and manipulators. OROCOS introduced the concept of using component-based architecture for robotics by basing its implementation on the Common Object Request Broker Architecture (CORBA) [72]. Moreover, the OROCOS project tried to push good practices for robotics, by creating a modular toolchain promoting interoperability between frameworks, separation of concerns [24], and component libraries [23].

The design choice of OROCOS of using CORBA and a component-based approach proved to be successful, today the framework is still used in application where hard real-time is a requirement. However, implementing real-time software is a difficult task not needed for a fair amount of robotic application. This spawned new approaches based on OROCOS and CORBA with the aim of supporting a wider variety of robots. Orca [17] implements a component-based design originally based on CORBA and then migrated to the Internet Communication Engine (ICE) [56]. The focus of Orca is on component re-usability, by defining a set of commonly-used interfaces and by providing libraries with high-level APIs. Another framework born from the premises of OROCOS is the Robot Construction Kit (RoCK) [59]. RoCK si built on top of OROCOS, it provides a tool called oroGen that can be used to automatically generate skeleton of OROCOS components. RoCK tries to use an automatic programming approach to ease the implementation of real-time component while maintaining intact their properties. An interesting feature of this framework is the focus on separating the problem-specific implementation from the framework-related code. Developers use oroGen to create the skeleton of the nodes, but they implement the functionalities in separate libraries that are connected at deployment time.

Given the variability of robotic platform, plenty of different frameworks and middleware specialising on a specific aspect exists. Yet another Robot Platform (YARP) is a middleware which

focus on providing a peer-to-peer architecture supporting an extensible variety of protocols (e. g., TCP, UDP, multicast, MPI, XML/RPC, etc.). It is used mostly in research and academia for humanoid robots requiring fast and reactive control loops.

DEVELOPMENT TOOLS

MODEL-DRIVEN APPROACHES

3 BACKGROUND

Nicholas of Morimondo: "We no longer have the learning of the ancients, the age of giants is past!"
William of Baskerville: "We are dwarfs, but dwarfs who stand on the shoulders of those giants, and small though we are, we sometimes manage to see farther on the horizon than they."

— The Name of the Rose, Umberto Eco

Since the beginning of this work, the aim was always to avoid "reinventing the wheel". By doing our initial analysis of the state of the art in robotic software development, software engineering, embedded system, modelling languages and formal methods, we realised that most of the necessary tooling was already available. The effort was in combining the existing technology to produce a result that is greater than the parts. Our target was to enhance robot software development by integrating methods, techniques and technologies from other fields.

This section provides to the reader all the necessary background to understand the technologies and the tools that will be used in this work. First, we present the Robot Operating System by giving an overview of its main elements, functionalities and structure. Then we move to modelling languages. The first one is the Architecture Analysis and Design Language (AADL), the main modelling language that we will use to design and describe robot architectures. We present the main components and their significance to our work. Moreover, we provide graphical and textual examples on how to represent basic AADL elements.

Contents

Figure 3.1: The "ROS Equation". It shows the key element composing the ROS environment.

3.1 ROBOT OPERATING SYSTEM

The Robot Operating System, as the name suggests, is an open source, meta-operating system (or middleware) designed to develop robot software. By creating a meta level on top of the operating system, ROS provides a series of functionalities to implement its distributed architecture, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running software across multiple platforms.

The main goal of ROS is to support code reuse in robotic. This is achieved by following the philosophy of giving great freedom to the developers and support them by providing a robust and flexible communication architecture, by developing a collection of tools and by creating a thriving community. Figure 3.1 perfectly synthesize what ROS is, it is not a framework, it does not provide libraries, guidelines, or developing rules. ROS is a communication infrastructure, a distributed network of processes to be individually designed and loosely coupled at runtime. A set of tools, ROS provides multiple interfaces to inspect (e. g., `rviz`, `rqt_graph`) the system, log (e. g., `rosbag`) the output, organise (e. g., `roslaunch`) the components, visualise (e. g., `rqt_plot`, `image_view`) data streams, navigate (e. g., `rospack`, `roscd`) the file system, build (e. g., `catkin`) the environment, and more. A repository of already available software modules implementing functionalities from low-level device drivers, to high-level planning, navigation, or manipulation algorithms. Lastly, ROS is the engaging community created thousand of developer around all world working toghether to

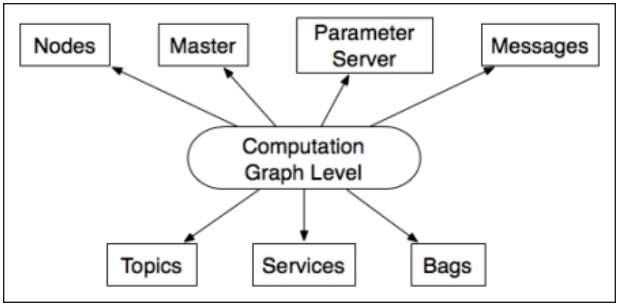


Figure 3.2: The ROS Computation Graph.

develop ROS packages and sharing them using the ROS Wiki[1] and supporting each other through ROS Question & Answers[2].

COMPUTATION GRAPH

Central to the structure of ROS is the Computation Graph (Figure 3.2), it represents the abstract infrastructure connecting all the elements of the ROS middleware. It should not be confused with the runtime graph composed by the currently active elements of a ROS architecture, which is just one of the possible instances. The Computation Graph consists in a peer-to-peer network where all the hypothetical ROS applications can connect to process, share, and exchange data, it defines the backbone of any ROS-based architecture. The main conceptual components connected to the abstract Computation Graph are nodes, as data generators and processors, the Master, as coordinator and name server, the Parameter Server, as parameter centraliser and provider, messages, as main form of data exchange, services and topics, as the prime communication channels, and bags, as pure data consumer.

- Nodes. They processes that perform computation, they produce, process and consume the data circulated in the graph. The modular design of ROS is based on the fine-grained functionalities implemented by the nodes. Each of them is

1 http://wiki.ros.org/
2 https://answers.ros.org/questions/

in charge of a specific subsystem of the robot (e. g., a sensor, an actuator, localisation, planning, etc.).

- Master. The role of the master is to provide a name registration service and a lookup system to the rest of the Computation Graph. It mediates the communication between nodes by initiating the connection between them.
- Parameter Server. ROS provides a centralised location to store all the parameters of the nodes. Parameters can be global (e. g., `/use_sim_time`) or defined in the namespace of a node (e. g., `/talker/frequency`).
- Messages. They can appear in various forms depending on the specific protocol used, however, all the communication in ROS happens trough the exchange of messages. A message is simply a data structure, comprising typed fields (e. g., integer, floating point, boolean, etc.). Messages can include arbitrarily nested structures and arrays.
- Topics. They represent the asynchronous communication system implemented by ROS based on the publish/subscribe paradigm. Topics are named channel, and nodes can subscribe to them to receive messages, or publish on them to produce messages.
- Services. They implements the ROS version of a synchronous communication system based on the client/server paradigm. Differently from topics, they are not a named channel, but remote functionalities identified by a name. Communication happens through an exchange of two messages: a request and a response.
- Bags. Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

Each ROS architecture represent a runtime instantiation of the Computation Graph where vertexes are represented by nodes and the other active elements (i. e., Master, Parameter server and, if

active, the bag recording system), and edges are represented by communication systems (i. e., service and topics). The existence of messages in the runtime version of the Computation Graph is transient, since they are generated, exchanged and then consumed. However, when they exist, they form a temporary ternary relation with the node and the communication channel.

While the Master acts as a coordinator to initiate the communication between nodes, after it is established they are connected to each others directly. For example, nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS, which uses standard TCP/IP sockets. This architecture allows for decoupled operation, where the names are the primary means by which larger and more complex systems can be built. Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library supports command-line remapping of names, which means a compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

For example, to read the measurements produced by a Hokuyo laser range-finder, it is possible to use the *hokuyo_node* driver, which interfaces with the hardware component and publishes *LaserScan* messages on the `/scan` topic. To process that data, a developer might implement a *laser_manager* node that subscribes to messages on the `/scan` topic. After subscription, the new node would automatically start receiving messages from the laser. The two sides are completely decoupled. All *hokuyo_node* does is publish laser range-finder measurements, without knowledge of which node will consume the messages. The only thing *laser_manager* has to do is subscribe to `/scan`, without knowledge of whether the topic is active or which node is generating the messages. The two nodes can be started, closed, and restarted, in any order, without inducing any error conditions. Later, it may be necessary to add a second laser range-finder to the robot, this requires to reconfigure the architecture. This can be done easily by remapping the names of nodes and topics. Two instances of the *hokuyo_node* are now necessary, and each of them can have its own name to

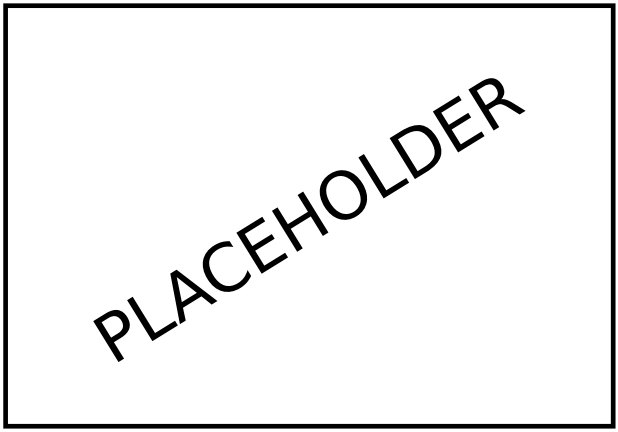


Figure 3.3: TODO

coexist in the graph as two different vertexes. Moreover, since they provide two different measurement, each `/scan` topic can be renamed accordingly (e. g., `/left/scan` and `/right/scan`). For the *laser_manager* to reach one of the topic, it is possible to follow the same remapping procedure, where the `/scan` topic is renamed to match the output of the driver.

COMPONENTS

In the Computation Graph the main active elements (i. e., performing computations) are nodes, they implements directly the functionalities of the robot and appears in the graph as a multitude of instances. However, other actors exists that perform small, yet crucial roles in the correct execution of a ROS architecture. The Master as communication mediator, the Parameter server to configure the nodes and the environment, and, even though it is not in the main concept of the Computation Graph, the Transformation Frame, a centralised system handling coordinate transformations in the robot.

MASTER – It exists as a single instance in the runtime graph, and it must be run before every other element of the architecture

using the command `roscore`. This is necessary because the Master acts as a name service, when a node is started it is registered to the Master, the same happens every time a new topic, both as a subscriber or publisher, or a new service, both as client and server, is created. By communicating with the Master, nodes can receive information about other registered nodes and their advertised topics and service, and, eventually, create the necessary point-to-point connections.

The communication between the Master and the nodes happens using a XML Remote Procedure Call (XMLRPC) protocol, which is an HTTP-based protocol that does not maintain an active connections. Nodes interact with the Master when they need to register a new communication or when they need to retrieve registration information. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run. The non-permanent connection combined with the lightweight XMLRPC protocol make it possible for the Master to manage very large and complex environments.

Figure 3.3 shows an example on how the Master mediates the connection between two nodes. At the beginning there are two independent nodes. A typical sequence of events starts with the *camera* node notifying the Master that it wants to publish *Image* messages on the topic `/images`. The combination of node and topic is registered in the Master, but since there is no subscriber yet, no data is actually sent. At some point, without any knowledge about the existing nodes or topics, the *image_viewer* node register in the Master a subscriber for the topic `/images`. Since the topic has now both a subscriber and a publisher, the Master can notify the two nodes, so they can establish a direct connection and exchange messages. From this point the Master is not involved in the communication any more, until one of the two nodes shuts down and its topic information is unregistered.

PARAMETER SERVER – As for the Master, only one instance of the Parameter Server exists in the system. This is not the only trait they share, since both are started together with the `roscore` command. The Parameter server is a shared, multi-variate dictionary that is

accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

As mentioned before, names have an important role in defining a ROS architecture. The Parameter server follows the same convention used when naming topics and nodes. This means that ROS parameters have a hierarchy based on the namespace introduced by the other elements of the graph. In practice, parameters can be accessed globally when using the full definition of their names (e. g., `/camera/right/exposure`), while are resolved depending on the current namespace when using a partial name (e. g., `right/exposure`). The hierarchical scheme also allows parameters to be accessed individually or as a tree. For example, let us consider a system where two parameters are exposed in the same namespace: `/camera/left/exposure` and `/camera/left/resolution`. It is possible to access each parameter directly and get their individual values or use a reference to the namespace (i. e., `/camera/left`) to receive a dictionary containing all the parameters existing in that namespace.

NODES – They are the main executable element active in the graph. Multiple different nodes and multiple instances of the same node can be running at any time. They represent the variability of the system, since their implementation and topology is defined by the developer. The aim of the nodes is to implement fine-grained functionalities of the robot, for example they can implement a single device driver, or a velocity controller, encapsulate a planning algorithm, or a localization system. ROS gives absolute freedom to the developer in the design and implementation of the nodes, however, to interact with the Computation Graph, few standard interfaces are defined.

To interact with topics, nodes implements publishers and subscribers. A publisher needs to be registered on a specific topic, and then can be called at any point during execution to generate a specific message and circulate it in the graph. Subscribers ref-

erence a topic and are bound to a function as a callback. When a new messages is detected on the topic, the callback is executed providing the instance of the message as parameter. Interaction with services is similar, it happens using clients and servers. To advertise a service a node needs to implement a server and bind it to a callback. The callback is triggered by a request message, and it is expected to return a response message with the result of the computation. To use a service a nodes implements a client. After binding it to the correct service, the client can be invoked at any time. While the remote service is in execution, the client node is locked waiting for the completion. A similar approach (i. e., client and server) is used for action, with the significant different that the execution is non-blocking.

There is only one thing a process needs to do to be part of the runtime graph and be considered a node: register itself to the Master. This single requirement encompasses the philosophy of ROS of giving basically no restriction to the developer. While not required, there is another element that characterise a ROS node: the spinner. The ROS spinner is the main loop of the component, it polls all subscribers, publishers, clients, servers to detect any new message or request. ROS provides few option when implementing it.

- *single spin*. The developer controls the frequency of the polling by invoking the spin command when necessary. It is useful when implementing a task that has to be repeated periodically (e. g., fixed frequency multiplexer).
- *single-thread spin*.The main execution thread of the process is locked in the polling activity, until a new messages is received by a subscriber or a server. This approach is used when the action of the node are purely reactive (e. g., a control component waiting for set-points).
- *multi-thread spin*. As before, the process is locked when the spinner is active, but instead of sequentially switching the execution between the polling activity and the callbacks, multiple threads are used, providing a variable level of parallelisation. This approach has to be used when the node

implements long callback with a fixed frequency (e. g., low-level sensor driver).

- *asynchronous spinner*. This is an alternative implementation of a multi-thread spinner. In this case the polling activity is not blocking, since it is implemented in a separate thread, and each callback is managed in a new thread. When implementing an asynchronous spinner, the developer has to be aware of all the potential concurrency problems introduced. Asynchronous implementations are useful when the node has to implement fixed-frequency callbacks while maintaining control on the main thread (e. g., sensor driver with independent communication threads).

The decentralised processing architecture implemented by ROS nodes provides several benefits to the overall system. There is additional fault tolerance, since an unexpected shutdown of a node may compromise the functionality of a subsystem, but not of the entire robot. With respect to a monolithic system, code complexity is significantly reduced, since the implementation is distributed in the single node and all the coordination and communication activities are managed by the Computation Graph architecture. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

ROS is designed to be a meta-operating system, and its focus is on accessibility, component reuse and hardware abstraction. Therefore, differently from other robotic frameworks, it does not provide a predefined structure for nodes and it leaves freedom to the developer when implementing them. The ROS-specific functionalities, publish and subscribe to topics, invoke and offer services, access the Parameter server, are all provided by a thin implementation layer called ROS Client Library. It is a collection of implementations, libraries and APIs that assist the developer in developing ROS nodes. Perfectly in line with the flexibility promoted by ROS, such libraries can be implemented in any programming language, since they need to implement general protocols like XMLRPC and TCPROS (i. e., ROS transport layer

based on TCP/IP). Currently, there are three main client libraries, with a particular focus on C++ and Python.

- *roscpp*.The C++ implementation of the ROS Client Library. Given the language of choice, this library is designed for efficiency, high execution speed and robustness. It is the most widely used library and it should be used when targeting the final deployment of a ROS-based architecture.
- *rospy*. This is the version of the ROS Client Library implemented in Python. It is aimed at providing advantages of an object-oriented scripting language, namely reduce development time and provide implementation flexibility to promote rapid prototyping and testing. Moreover, it is ideal for non-critical-path code, such as configuration and initialization code
- *roslisp*. While actively supported, this implementation of the ROS Client Library in LISP exists mostly for legacy reasons. It is used for the development of the planning libraries.

TRANSFORMATION FRAMES – When describing the physical structure of a robot, the position of each element has to be defined with respect to the others. For example, to estimate the position of a mobile robot though vision, it is necessary to know where the camera is located on the robot, to estimate then the expected position of some reference points and translate it back to the base of the robot. The more degrees of mobility a robot has the more complex this description become, Figure 3.4 shows all the possible coordinate frame in a THORMANG3 robot. Each joint of the robot can move with respect to the previous one, therefore every point on the robot and in space can be defined with respect to a multitude of coordinates system, since each of these systems defines a new rigid transformation. The complexity of managing all the coordinate systems grows dramatically every time a new one is defined, and the number of total frame may increase dynamically during execution, since any new interactive element in the environment (e. g., an obstacle to avoid or an object to grasp) may create a new one. Given how central coordinate frames are

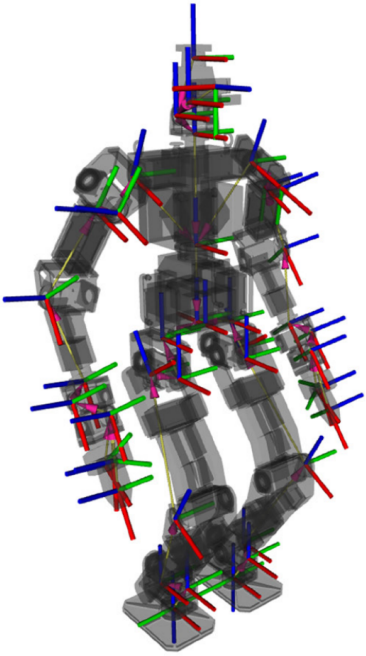


Figure 3.4: A graphical representation of all the coordinate frames necessary to completely describe the structure of the THORMANG3

when developing a robot, ROS provides its own set of tools to manage them: *transformation frames*, commonly referred as *tf*.

Of course, *tf* cannot replace the designer in defining the correct reference frames for each element of the robot. However, what it does is to create a distributed system to manage coordinates that free the developer from worrying about conversions between one and another. Thanks to *tf*, at any moment in time, the developer can define a location in space in a specific coordinate system and easily convert it in a different one. To achieve this, it is necessary to maintain a consistent transformation tree, an example is shown in Figure 3.5. *tf* provides conversion from any frame to another (e. g., from `map` to `right_wheel`) at any point in time, but only if a

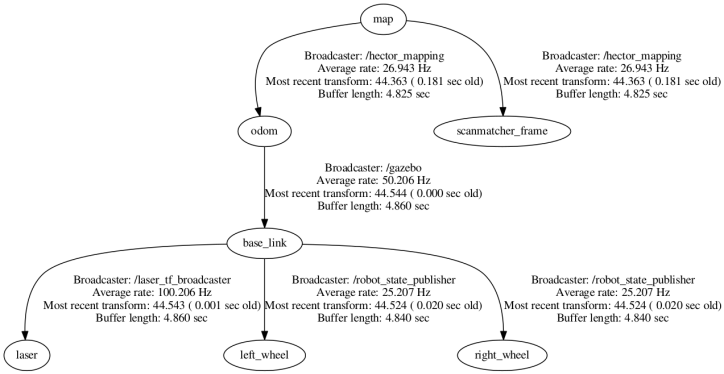


Figure 3.5: An example of a complete tf tree for a mobile robot.

complete chain of transformation between the two frames exist in the correct time frame.

To maintain ha consistent transformation tree, the developer can use the many accessory tools provided. When a coordinate system does not move with respect to its parent reference frame (e. g., the position of a sensor on a robot), it is considered a static transform. They can be broadcast using the *static_transform_publisher* node, which can be configured to create a latched publisher that provides an up to date static transform every time is requested. A transform is dynamic when it changes its relative position in time (e. g., the position of the robot with respect to the map). In this case, to update the transformation tree, it is necessary to use the APIs provided by *tf* during the execution of the node responsible of estimating the dynamic transform. The node has to declare a *tf* broadcaster, then, at any time during the execution, can be used to notify the updated version of the dynamic transform to the system. Since *tf* can estimate the chain of transformations at any point in the past, it is important that each new dynamic transform is broadcast with the correct timestamp.

When a node needs to query *tf* to request a specific transform, it can do that by declaring a *tf* listener. The listener can be used to receive any kind of coordinate frame at any point in time, by specifying during the invocation a parent frame, a child frame

and a timestamp. While there is no central location containing all the coordinate systems, and *tf* uses a distributed architecture to store and share them, most of the actual computation happens in a "transformations buffer" embedded inside the listener. This means that while, theoretically, it is possible for *tf* to unravel the entire history of a coordinate system, in practice this is limited by the buffer created by the specific listener. However, present-time transform are always available at any depth of the transformation tree. In summary, *tf* is one of the most powerful tool provided by ROS, since it removes a significant burden to the developer when designing and implementing complex interconnected robotic systems.

LAUNCH FILES – The Master, together with the Parameter server, are run by the `roscore` command, nodes are activated using the `rosrun` command and *tf* provides a special command to broadcast static transforms. When an architecture grows, so does the complexity of executing all its elements in the correct order and with the right configuration. In order to alleviate this burden ROS provides a system called `roslaunch`.

A launch file processed by `roslaunch` is an XML file containing a list of the elements that needs to be run for a specific architecture. The Master and the Parameter server are run automatically when executing the `roslaunch`, hence by writing a complete enough launch file it is possible to execute the entire architecture with a single command.

A launch file is highly configurable. Using the XML tags, a developer can: rename nodes, remap topics, provide command line configuration, include complete parameter profiles through YAML files, define global parameters, enable node restart and include other launch files. The potentially hierarchical structure of launch files can be used to obtain a partial description of the runtime topology of the architecture, however, currently there is not tool in ROS to provide this kind of visualisation.

COMMUNICATION

As any graph, the ROS runtime graph is composed by vertices and edges. In the previous section, we discussed the main processing elements, the vertices, while in this section we cover all the communication protocols, the edges. One of the main characteristics of ROS is its distributed architecture, nodes can be executed on different machines, but the complexity of transmitting messages through multiple physical mediums is hidden by the middleware functionalities. Communication in ROS happens mostly through a very flexible protocol based on a publish/subscribe approach: the topics. While this one-size-fits-all approach works for most situation, ROS also provides alternative solutions like services and actions.

MESSAGE – Central to the communication system of ROS is the concept of messages, since they are used, in different forms, by all protocols. Nodes communicating through topics exchange messages directly, the client/server interaction implemented by services is achieved by a pair of messages, while actions use a more complex system that relies on a triple of explicit messages combined with hidden messages used by the protocol.

In every aspect of its design, ROS promotes simplicity, thin approaches and a straightforward implementation, messages are no exception. They are simple data structure, composed by constants or typed fields. The field types can be:

- one of the standard built-in types. They are the common types usually found in programming languages. The available types are: boolean, a logic value that can be *true* or *false*, integer, there are signed and unsigned options with different sizes (i. e., 8-bit, 16-bit, 32-bit or 64-bit), floating point, for any non-integer number, available as 32-bit and 64-bit, string, any ASCII string, therefore Unicode characters are not supported, time and duration, represented as a pair of seconds and nanoseconds;
- other messages. ROS supports a nested structure for message typing, this means that any other message type, predefined in ROS or custom-defined by the developer, can be

used as a type in field in a message. This promotes reuse of standard, and previously defined, messages, and a hierarchy of concepts. For example, it is natural to assume that a *Polygon* is defined as a list of *Point*, or that a *Pose* is composed by *Position* and *Orientation*;

- any of the previous type can be arranged in a fixed-length array or in a dynamic list. Together with the nested structure, it makes the design of ROS message simple yet powerful;
- the special *Header* type. While it is defined as a message in the package *std_msgs*, the *Header* is unique since it can be referred directly without specifying the package and it is meant to exists as an unique field on the root of the message to provide a generalised ID. The *Header* type has three fields: `seq`, an unsigned long integer designed to be an increasing identifier of the message instance, `stamp`, the timestamp of the creation of the message, and `frame_id`, a string containing the name of the frame of reference associated with the message.

Messages are defined using a very simple message definition language. Each field is described by a pair of keywords, one defining the type and the other defining the name. To define an array or a list, it is possible to add two square brackets near the type, eventually specifying the size (e. g., `int32[]` or `Point[3]`). Constants are defined in the same way as fields, except that it also assigns a value using the equal sign (i. e., `=`). No other data structure is allowed in the message definition, to nest type it is necessary to define them in a different message and include them as types. Multi-part messages used in services and actions are defined following the same rules, each part is separated by the others using "`---`" as a separator.

TOPIC – The main communication system of ROS is based on an asynchronous protocol implementing an anonymous publish/subscribe paradigm. In practise, topics are named buses that nodes can use to exchange messages. Given the publish/subscribe paradigm, combined with the anonymity of the message-based

communication, the use of topics decouples the production of information from its consumption. Without any explicit information in the content of the message, nodes are not aware of who they are communicating with. Instead, nodes that are interested in a specific data stream subscribe to the relevant topic, while nodes that generate data publish to the relevant topic. A for many other design choices, ROS is very flexible and permissive in the structure of topics, hence, multiple publisher and multiple subscribers can read and write from the same topic at the same time.

Topics are intended for unidirectional, streaming communication, therefore, they are not supposed to provide a reliable communication. Topics do not guarantee the delivery of the messages nor have an expected delivery time. While the communication is mostly reliable when the two connected nodes are executed on the same machine, the potentially distributed nature of a ROS architecture (i. e., nodes running on different machines) combined with the unknown physical communication medium force the developer to treat ROS topics as a communication channel subject to potential data loss.

While there is no limit on the number of publishers and subscribers connected to a topic, they all need to write or expect the same message. Topics are strongly typed by the ROS message type used to publish to them, and nodes can only receive messages with a matching type. The Master does not enforce type consistency among the publishers, however new publisher with a mismatched type faces a communication error when trying to write on the topic. On the contrary, the Master will block subscribers from establish message transport if the types do not match. Lastly, all ROS clients implements a client-side check to make sure that an MD5 computed from the message file matches the signature of the topic. This check ensures nodes are compiled from a consistent code base.

To implement topic communication, ROS currently supports message transport based on both TCP and UDP. The TCP/IP is the default transport, known as TCPROS, and streams message data over persistent TCP/IP connections. Since it is the default, TCPROS is also the only transport that ROS Client libraries are required to support. The UDP transport, which is known as UD-

PROS, is currently supported only by the C++ ROS Client Library (i. e., *roscpp*). UDPROS is a low-latency, lossy transport, hence is best suited for non-critical tasks requiring high-speed communication and faster response (e. g., teleoperation). ROS nodes negotiate the desired transport at runtime, therefore, a node implemented to use the UDPROS transport can always fallback on TCPROS if the destination node does not support it. This negotiation model enables new transports to be added over time as compelling use cases arise.

SERVICE – The publish/subscribe communication provided by topics is extremely flexible and suitable for a variety of situations in a robotic architecture. However, its *n-to-n*, one-way, asynchronous transport, combined by the implicit unreliability of the communication, is not appropriate for all interactions. In distributed systems it is often necessary to implement synchronous, bi-directional, reliable interactions, in ROS this is achieved using services. They implement a remote procedure call (RPC) based on a client/server paradigm. Services are defined by a pair of messages: one for the request sent by the client, and one for the reply sent by the server. Differently from topics, the name of the service advertised by a node does not represent the communication channel, but an entry on the Master. In practices, it means that differently from a topic, it is not possible for an external entity to "listen" to a service communication.

Service calls are one-shot interactions. A client start the communication by requesting to the Master who is exposing a specific service. After that it establish a direct connection with the server and send the request message, after the service-specific processing, the server sends back the appropriate response and the connection is closed. Multiple server can expose the same service (i. e., advertise a service with the same name), there is no way for the client to pick a specific server, and the Master will provide one of all the available. In extreme cases, this means that successive calls of the same service will be answered by different servers. To avoid this, a client can make a persistent connection to a service, which enables higher performance at the cost of less robustness to service provider changes.

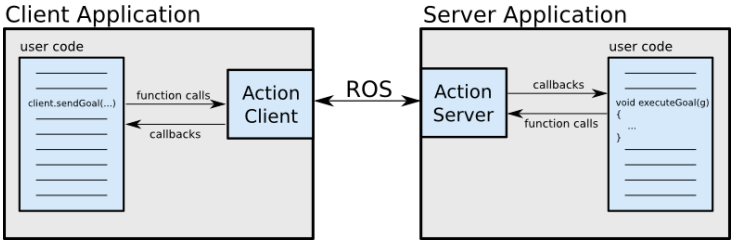


Figure 3.6: Communication interface of the ROS actionlib.

As for topics, services are strongly typed by the service message they use. In this case is the server that imposes the type of the communication, since it is the one exposing the service. Clients are not allowed to establish a communication if the requested type does not match the server. As an extra layer of consistency, services are versioned by an MD5 sum of the service message file. Nodes, both implementing server and client, are only allowed to start a service interaction if the the MD5 sum matches.

ACTION – Services implement a synchronous client/server protocol, this means that after sending the request, the client needs to wait until the server provides the response. This approach is suitable for short interactions, when the waiting time is not detrimental to the correct execution of the client, however, it is incompatible with requests that trigger long multi-step procedures. The ROS answer to an asynchronous client/server protocol is the *actionlib* package, or "actions".

Actions realises their asynchronous protocol by using a collection of topics, this "under the hood" implementation is hidden to nodes by providing two interfaces: an *ActionClient* and an *ActionServer*. The client and server then provide a simple API for developer to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks. Since action are designed to trigger complex and long activities, they implement a system to track the evolution and the current status of any active task, and to interrupt or pause any active action.

As said before, the actual communication is implemented through topics, to fulfil all the expected functionalities, the action protocol

uses five different communication channels. Two of them are from the client to the server: goal, used to activate the action and send the current goal, and cancel, used to interrupt an active action. The remaining three are from the server to the client: status, used to notify clients on the current state of a goal (e. g., active or paused), feedback, used to send periodic auxiliary information about a task, result, used to send to the client a one-time information upon completion of a goal.

While not all topics are used directly, to implement the action protocol, the developer needs to define an action message file. The file is composed by three different parts, each one representing a different phase of the protocol: activate the action, receive updates and receive the result.

- Goal: this field represent the final goal of the task and it is used to activate the action. It is the asynchronous counterpart of the service request, and as for the request it is sent by the *ActionClient* to the *ActionServer*. For instance, for a complex navigation task, the goal would a *PoseStamped* message containing the destination of the robot.

- Feedback: this field provides incremental updates on the current status of the goal. It is unique to the action protocol and it has no counterpart in a service. This message is periodically sent by the server to the client. In the case of the navigation task, a feedback message would be the current position of the robot along the path.

- Result: this field return the result of the task and represent the end of the action. Its synchronous counterpart is the service response provided by the server. A result message is sent only once by *ActionServer* to the *ActionClient* upon completion of the goal. In the example of a complex navigation task the result is not particularly meaningful, however, it is significant, for instance, when the task of the action is to calculate a complex behaviour for the robot. In this case, the result message contains the resulting behaviour.

FILESYSTEM

Given its nature as a middleware and meta-operating system, additionally to all the communication and execution functionalities, ROS also provides a filesystem on top of the original provided by the native OS. It is possible to navigate this ROS filesystem using a series of commands, for instance, `roscd` to move to a specific folder, or `rospack depends` to list all the dependencies of a specific component, and many more.

Central to the design of the ROS filesystem is the concept of packages. A package might contain nodes, a ROS-independent library, a dataset, configuration files, a third-party software, or anything that can be logically considered part of a module. The goal of these packages is to provide useful functionalities in an easy-to-consume manner, to promote software re-usability. In general, ROS packages follow a "Goldilocks" principle: enough content to be functionally complete, but without overloading the package which become heavyweight and difficult to manage.

The build process of ROS is atomic with respect of packages, this means, for instance, that is not possible to build a standalone node, or a loose message. This approach is connected to the naming system of ROS, where any entity (e. g., nodes, messages, services, etc.) is identified by a combination of entity name plus container package name. Being the atomic unit of build, a packages is also the target container for release and deployment.

Packages are easy to create by hand since the only requirement is to include a *package.xml* file. However, tools exist to support this procedure, like `catkin_create_pkg`. While only one file is required to define a package, usually they follow a standardised structure:

- `include/pkg_name`: folder dedicated to C++ include headers;
- `msg`: here all the *msg* files used to declare messages are contained;
- `srv`: service counterpart of the `msg` folder, it contains all the service definition files;
- `src`: main folder containing the node implementation, it is usually dedicated to the C++ source files;

- `scripts`: secondary folder for the node implementation. It contains all the source files that do not require compilation, it is usually dedicated to the Python implementation;
- `CMakeLists.txt`: CMake build file;
- `package.xml`: the only mandatory file, it contains package specifications and meta-data.

3.2 ARCHITECTURE ANALYSIS & DESIGN LANGUAGE

The Architecture Analysis & Design Language is a very powerful modelling language designed to capture the architecture of embedded systems by using architectural models that provide a well-defined and semantically rich description of the runtime architecture. This description encompasses multiple aspects of the system: hardware components, to encode the underlying physical layer of the system, software components, to define the runtime behaviour of the architecture, the interaction between them, for example deployment of software on specific hardware and communication between different execution units, and the defining properties of each modelled element, to better characterise any particular system.

In AADL, components are defined using a dichotomy between specification and implementation. The component type declaration is used to define the category (see Table 3.1) and the interfaces (i. e., features) of the component; this correspond to a specification sheet that provides a description of the component as a black box. For a specific type it is possible to define multiple component implementation declarations, each of them defines internal structure of the component (i. e., subcomponents and their interactions). This is equivalent as defining multiple blueprints for building a component from its parts, each of them a possible implementation of an already defined specification. To specify even more the characteristics of a component, especially its runtime behaviour, it is possible to use properties. AADL already provides a collection

of predefined properties, and more are available by including standard annexes for specific analyses, moreover, an user can defined his own properties by defining additional properties sets. Together, all these declarations (i. e., type with implementation with a set of properties) define a pattern for a component, which are referred as component classifiers.

Component types and implementations are defined and organised using packages; they are, essentially, libraries of component specifications that can be used in multiple architecture definitions. Packages have public and private sections to support information hiding. The public section of a package contains all the specification that will be available to other packages, while the private section can be used to hide the specific component implementation. In AADL, everything is organised in packages, an exception are property sets. They are special container for user-defined properties, they act like packages and can be imported in other definition, but only properties can be defined in properties set.

To model a full architecture it is necessary to first define all the necessary component classifiers, or import the existing ones in previously defined packages. In the case of a robot, for example, it is necessary to define the physical sensors as devices and the execution platform as a combination of processors, buses and memories. On the software side, the designer could import previously defined software component as processes or define more in new packages and then import them. After this initial definition, a complete architectural description is created by integrating in a fully specified system implementation instances of the previously defined component classifiers. This hierarchy represents all the interactions between components and the architectural structure of the modelled system. These interactions cover multiple aspect of the system, they encode the communication between components through data and events, and the physical connections between them. They also capture the assignment of software to hardware (e. g., on which physical processor or processing unit a specific process will be executed). The full model of the system under analysis is obtained by instantiating this top level system implementation. This instance model can then be used to analyse operational properties of the system, ranging from syntactic

CATEGORY	DESCRIPTION
	Application software
process	Execution unit with a protected address space.
thread	A schedulable execution path.
thread group	An abstraction to logically organise threads.
data	Abstraction for data units.
subprogram	Callable sequentially executable code. It represents call-return functions.
subprogram group	An abstraction to logically organise subprograms.
	Execution platform
processor	Schedule and executes threads and virtual processors.
virtual processor	Logical resource that can schedule and executes threads. It must be bound to one or more physical processor.
memory	Stores code and data.
bus	Interconnects processors, memory and devices.
	Composite
system	Integrates software, hardware and other system components.
	Generic
abstract	Define a runtime neutral component that can be refined into another component category.

Table 3.1: Component categories.

compliance and basic interface data consistency to assessment of quality attributes and behaviours.

The key characteristics that make AADL suitable for our approach are the inheritance between components and the possibility to use partially defined components and interfaces that can be refined later in the design process. In practice, inheritance exists as a form of extension of existing components. A new classifier (i. e., component type and implementations) can be defined by extending an existing one; the extended classifier inherits all the characteristics of the base one: interfaces, subcomponents, properties, internal connections and modes. The extension declaration can be used to refine the new classifier by adding new elements, specifying existing elements inherited from the base classifier, restricting subcomponents to a specific mode, completing the definition of partially defined sections.

Partial definition is achieved in two ways: by using abstract components or by exploiting prototypes. The *abstract component* is a generic category that can be used in place of any other component type or implementation without having to specify a runtime category. A model with an abstract component cannot be instantiated, however they are extremely useful to define the initial conceptual description of the system during an iterating design process, or architecture templates and patters that can be used as reference libraries by designers. The *prototypes* act as placeholders for classifiers and they can be referenced anywhere a classifier would normally be referenced. The actual classifier can be specified later when referencing the parametrised component, e. g., when extending the classifier or when declaring a subcomponent. Prototypes are useful to create reference architectures or configurable product line families by providing, essentially, a parametrised classifier template that a designer can easily specify while following the structure already provided. An example is the data type exchanged between two components; the template of the component define the existence of the communication channel, but it uses a prototype for the actual type of the data. Because of the prototype, the designer needs to define a data type in order to be able to instantiate the model, but there is no restriction of the original definition of the template.

AADL is a formal declarative language described by a context-free syntax. This well-defined semantics is a key aspect of the language and a strong advantage, especially for quantitative system architectural analysis. Textual AADL is the main, more straightforward and detailed way to interact with the language, however, there are standard graphical representation that correspond to the textual definition. During the design of an AADL model, either of both representation can be used, a good strategy is to first define the skeleton of the model graphically, and then finalise the description using textual AADL. This process is supported by the Open Source AADL Tool Environment (OSATE), in this development environment a designer can easily switch between one representation of the language and the other, and any modification is propagated in all representations.

In the reminder of this section, we present more in details the component categories of AADL relevant to our work. We provide a description of the logical meaning of each category and their interactions, to better justify how used them to model a robotic architecture.

SOFTWARE COMPONENTS

These categories are used to model the executable architecture of the system, they encompass functional units as processes, execution path as thread or thread groups and executable code such as functions, procedures and libraries as subprograms and subprograms groups. Moreover, the data category can be used to represent the application software artefacts, some examples are data types, configuration files, internal data structures and communication messages. In addition to the semantic provided by the category itself, additional information associated to runtime (e. g., dispatch protocol and frequency of a thread) and non-runtime (e. g., source code associated to a specific subprogram) can be specified using properties.

PROCESS – It represents an encapsulated execution unit; the address space, the persistent state and all internal resources are all protected and they are not accessible by external elements directly.

The internal functions of the process are exposed using different kind of ports and interfaces (i. e., features): event ports can be used to trigger a behaviour or data ports for communication. Syntactically, a designer could provide access to the internal persistent state of the process, but, logically, processes usually represent protected address spaces. The process category is, basically, just a container that defines an executable entity, therefore it doesn't include an implicit definition of a thread; this means that a complete process specification has to include at least one explicitly defined thread. The allowed subcomponent categories are: thread, thread group and data. Properties can be used to specialise the runtime behaviour of a process, for example it is possible to specify the source code associated with the process, or even the actual binary that will be executed.

THREAD – It represents an execution path through code, that could, potentially, be executed in parallel with other similar execution paths. The executable code modelled by a thread exists within the protected address space defined by the process container. Although the name of this category suggest a direct binding between the model of a thread and a physical thread on a system, conceptually, an AADL thread is more versatile. A thread can be implemented by a single operating system thread, or represent one of multiple logical threads mapped on a physical one. A thread may also represent an active object. Logically, an AADL thread revolves around the property of being schedulable; threads can be bound to processors or virtual processor and they have multiple properties to specify their scheduling behaviour. The possible values of the *Dispatch_Protocol* property cover the most common behaviour expected by a schedulable execution path.

- Periodic, a repeated fixed time interval dispatch with the assumption that the execution time is shorter than the period.
- Aperiodic, a port-based dispatch triggered by an external source, if the thread is still executing when a new dispatch arrives a queue based system is used.

- Sporadic, the dispatch is triggered by external events on a port, but a new dispatch cannot happen before a specific interval of time.
- Timed, thread are dispatched after a specific amount of time if no event triggers it before. Basically, it is an aperiodic dispatch with a time-out.
- Hybrid, this dispatch method combines a periodic and aperiodic. A thread is dispatched by an external event or after a fixed amount of time.
- Background, a thread is dispatched once and it is executed until completion.

A thread can exist only within a process or as a direct subcomponent or as part of a thread group. The possible subcomponents of a thread are: data, to capture a persistent local state, or subprogram and subprogram group, to model a local call to a functionality. The interaction between threads can happen through ports, by accessing shared data component at process-level or by calling a subprogram serviced by another thread.

THREAD GROUP – It can be used to organize threads within a process in a hierarchy when they are logically related or when it is necessary to create a encapsulated space with respect to the rest of the process. The unified frontier presented by threads in the same group can be used as a common interface when a designer wants to capture, at the same time, an high level functionality and the low level constituting elements. Other than thread and other thread group, the legal subcomponents are data, subprogram and subprogram group. All these subcomponents are directly accessible by the threads in the group, but are reachable by any external element only through ports.

DATA – It can be used to model any kind of data exchanged, saved or defined in the system. Data component instances can appear in three different forms: as data subcomponents to represent persistent data (e. g., the state of an object), included in data or event data port to specify the type of data exchanged

in the specific communication, as parameters declaration of subprograms. As subcomponents, a data component can have more data components, to model a record-like structure, or subprograms, to evoke the concept of a method associated to a specific data type or class. AADL is not a data modelling language, however provides enough flexibility to be used as such. The most reasonable approach is to use the data category to map all the information relevant to the model, and then exploit properties to specify a more detailed description of the data using a more suitable language.

SUBPROGRAM – It represents a callable unit of sequentially executable code. The subprogram type represent the signature of function, procedure or method modelled, while the subprogram implementation represent the internal functionalities. The implementation is not required to instantiate a model, however, if necessary, data components can be used to describe local variables and nested subprograms define the execution sequence. Subprograms support data access to access a shared persistent state or outgoing ports to model exceptions and errors; moreover, data components can be used to model parameters and the return value. There are two ways to model subprogram calls: by referring to the subprogram classifier, or by using a using a subprogram access feature. The first approach is used when referring directly to the subprogram (e. g., to specify the subprogram as the executable code of a thread), while the latter is used to model indirect calls (e. g., to model remote service/procedure calls or, in combination with a data component, to model an object oriented approach). Various properties of the subprogram can be used to specify the actual executable code to be used in the implementation (e. g., *Source_Name*, *Source_Text* and *Source_Language*), others are related to the calling and execution of the subprogram itself (e. g., *Allowed_Subprogram_Call* and *Compute_Execution_Time*).

SUBPROGRAM GROUP – It can be used to represent a collection of callable routines. For example, a subprogram group type models the API of a software library by using a series of subprogram accesses, while different subprogram group implementations can be used to model multiple implementations of the same library (e. g.,

different versions or implementations in different languages). The possible subcomponent of a subprogram group are: subprogram, to define the actual content of the group, subprogram group, to create a multi-level hierarchy, and data, to define a persistent state shared by all subprograms in the group.

EXECUTION PLATFORM COMPONENTS

These categories are used to model the resources of the computer system and the elements of the external physical environment. To model the physical resources of the system, a designer can use processor, bus and memory categories. Each of these categories represent the concept behind these physical system and not the actual object, so a processor can be a CPU, but also a processor board including operating system functionalities. In the same way, a bus component can be used to model a physical bus on a board, or a network connection such as Ethernet or CAN bus. Both these components have their virtual counterpart: a virtual processor can represent a scheduler or a virtual execution environment, while a virtual bus can model a communication protocol or a virtual channel. Memory components represent any kind of memory present in a system, from RAM to cache as well as persistent memory such as hard drives. To model sensors, actuator or physical elements of the system it is possible to use devices.

PROCESSOR – The definition of processor is related to the concept of thread. A processor represents the hardware and associated software that is in charge of scheduling and executing of threads. In practice, this category can be used to model both low-level hardware of an embedded system and the high-level platforms together with operating system services, depending on nature of the model and the system. To support this, memory and bus are possible subcomponents and they can be used to define the internal function of the execution platform. To correctly instantiate the model, a processor has to be associated with a memory, it can be internal as a subcomponent or external connected via a bus. The properties available can be used to specify the runtime characterist-

ics of the hardware (e. g., *Clock_Period*) or the physical description of the component (e. g., *Hardware_Description_Source_Text*).

VIRTUAL PROCESSOR – It represent the logical counterpart of a processor, it is a virtual resource for scheduling and executing software. It can be used to model any kind of virtualization platform (e. g., Java VMs, Docker containers, virtual environments), partitions of physical processors or hierarchies of schedulers. To instantiate a model a virtual processor has to be associated to a physical one, or as a subcomponents or by binding. Properties specific to this category are related to the binding between virtual and real processors, the others are the same of the processor category, with the exception of those used to describe the physical hardware (e. g., hardware description and clock properties).

MEMORY – It represents any kind of storage for data and executable code. A memory category can be used to model randomly accessible physical storage (e. g., RAM and ROM), reflective memory, or permanent storage. A memory component can be used as a subcomponent of a processor to model a complete execution platform, or can exist as independent in a system to define more complex architectures or shared memories. Typically, two types of software components are bound to memories: process and data. A process has memory requirements for code, static and dynamic data, while a data component bound to a memory represent persistent data shared between different threads. Properties can be used to define the physical characteristics of the memory, such as word and total size, base address and access protocol.

BUS – It represents the physical connection between hardware components and the associated communication protocols. Some examples of the type of connection modelled by the bus category are PCI, CAN, Ethernet and wireless network. Another use of this category is to represent physical resources distributed to multiple physical components such as electrical power. Bus can exists as a subcomponent to any other execution platform category (i. e., processor, memory, device), however, nested buses are not permitted; only a virtual bus is accepted as subcomponent. Properties

can be used to specify details about the physical connection (e. g., *Transmission_Time* and *Allowed_Message_Size*).

VIRTUAL BUS – It represents a logical abstraction of a communication channel, such as a virtual partition of a physical bus, communication protocols or hierarchies of protocols by defining dependencies between multiple virtual buses. Since this category can be used to represent protocols, it can be referred in other components properties (e. g., a processor specifying *Provided_Virtual_Bus_Class*) to specify their supported communication standards.

DEVICE – It represents entities that interface with or are part of the external environment, such as sensors (e. g., cameras, laser rangefinder, GPS), actuators (e. g., motors, valves, pumps), or peripheral I/O. A device component has a dual software and hardware nature, since, as an abstraction, it can be used to model the physical component together with its driver; this means that a device support both ports and subprogram accesses to communicate with software components and bus accesses to interact with hardware components. The subcomponents available for the device are used to better describe the interaction between the external element and the system; a virtual bus can be used to specify the protocol of the communication, a bus to model the physical connection provided and a data component to capture the type of the data exchanged.

COMPOSITE AND GENERIC COMPONENTS

System and abstract component categories are not directly associated neither to software nor to hardware components, they are used to define conceptual and generic constructs. They provide to AADL the tools necessary to support modular and reusable models, by aggregating component together and by providing partially defined interfaces that can be refined during successive design phases.

SYSTEM – It is an abstraction that represent a composite component (i. e., a container for other components). It can include software, execution platform or other system components with

no restrictions. This means that it is possible to create system containing only hardware components (e. g., a processor board), only software components (e. g., a software control system), a combination of software and hardware (e. g., a complete embedded system), or a combination of all these as direct subcomponents or as contained in other systems. Even the extreme case of a system consisting only of system components can be used as a generic representation of a component-based architecture. Given its nature as a container and aggregator, any type of component is an admitted subcomponent of a system. Although the aggregation defined by the system is only conceptual, it creates an actual frontier between the subcomponents inside and those outside; this means that any communication needs to go through features (i. e.ports and accesses) defined on the system.

ABSTRACT – It is a generic component category that can be used to declare a component type and implementation without specifying a specific category. By using this component as the only category in a system it is possible to create a conceptual component-based view of an architecture. Alternatively, by combining abstract and normal components in a system definition a designer can define a reference architecture to be specialised when necessary. Lastly, abstract components can be used to create partially defined components that act as libraries of design patters. Abstract categories can be refined in any other category, for this reason any component (software or execution platform) is admitted as a subcomponent. The same is true for properties, since the abstract category support every possible property. However, when an abstract component is refined to an actual category, only the properties and subcomponents admitted for that category are valid.

COMPONENTS INTERACTIONS

In AADL, there are multiple ways to define interactions between components. In the previous section, we described software and hardware categories and, by doing so, we introduced two basic form of interaction: subcomponents and bindings. The most

straightforward form of interaction is the relationship between components and subcomponents; models are described in a hierarchical way where higher level components are composed by lower level subcomponents and the designer can decide his own level of granularity for the architecture. For example, a system can be modelled down to the executable routines, but with no hierarchy definition in the middle (i. e., a subprogram in a thread contained in a process inside a system), or defined only conceptually by using a strict hierarchy of the components (i. e., system of systems containing only processes as subcomponents), or any intermediate combination. Components have direct access to their subcomponents at any depth, using a dot notation similar to object-oriented design (e. g., `system.process.thread.subprogram`). The concept of binding is specifically designed to relate software and hardware components. A memory component binds multiple data components to specify the physical location of the variables they model, or a process is bound to the processor that will execute it. In practice, this can be seen as the deployment of software on a specific hardware and it is fundamental to perform analysis of operational properties (e. g., performance, latency, fault tolerance), simulate the execution of the system and generate the build configuration.

The main form of interaction between different component is the use of connections. They are a very versatile form of interaction and they can represent a multitude of types of communications (e. g., remote function call, message passing, inter-process communication, variable access, interrupts), the differentiation of the communication protocol is achieve by defining externally visible features on components (i. e., interfaces on the frontier of a component). There are five different types of feature (ports, access, groups, abstract and parameters) and each of them has subtypes depending of the type of information exchanged (e. g., data or events) or the component they are defined on (e. g., bus access or data access). Since subprogram are used to model procedures, functions, methods or, in general, any callable routine, they support two unique form of communication: parameters and calls. The former, as the name evoke, defines the relationship

between data components and subprograms; the latter represents the sequence of execution of multiple subprograms.

PORTS – Ports are the most straightforward type of features. They are an interface for directional transfer of data, events or both into or out of a component. Compatible ports (see Table 3.2) can be connected to define directional pathways for such transfers between components. Ports are defined in the component type declaration and are specified by name, direction, type and a data identifier. The name has to be unique in the scope of the component and it can be used to recall the port, both inside (directly) or outside (via dot notation) the component to define properties and connections. There are three possible options for the direction of a port: `in`, to specify an input, a flow of data or events from outside the component, `out`, represents an output, data or events coming from the component, `in out`, a bidirectional port, both input and output, this type of port supports incoming and outgoing connections. Three different types of ports are supported, and they represents different type of communications between components. Event data ports are meant for asynchronous communications, for example messages exchange. They model asynchronously sending and receiving data and the presence of a queue to store unprocessed messages while the receiving component is busy. Data ports are similar, since they model data exchanges, but without a queue. They are sampling ports, they retain only the most recent arrival. In the definition of both these types of ports it is possible to specify a data identifier that model the nature of data exchanged in the communication. Event ports represents triggers for discrete events and they carry no data. They can be used to model all kind of external events, from low level hardware interrupts, to signals from the operating system.

DATA AND BUS ACCESSES – In a system there are multiple shared resources, for example memories, log files, communication channels, sensors, input and output devices. In AADL, most of these resources are characterised by two component categories: data and bus. To model the concurrent access of components to these shared elements it is possible to use access features. Two types of access features exist: data access and bus access. Their conceptual

From/To	`data`	`event data`	`event`
`data`	Yes	Yes	Yes
`event data`	Yes	Yes	Yes
`event`			Yes

Table 3.2: Inter-port compatibility.

definition and syntax is almost the same, the only exception is the use of the right keyword when referring to one or the other type. In the feature definition, they are identified by an unique name, that can be used to refer to a specific access in the model. Similarly to ports, accesses have a direction, but it is achieved by defining either a requires access feature, indicating that a component needs access to a shared resource, or a provides access feature, meaning that a component allows access to a shared resource defined as a subcomponent within it. Optionally, it is possible to specify, in the definition of the access, a component identifier referring to a specific data or bus classifier, depending on the type of access. As any other feature, path between accesses are created using connections, differently from other features, the chain of connections in and out components does not end in a feature, but it continues to the shared resource. Connection between access features can be bidirectional ($\leftrightarrow$) or directional ($\rightarrow$); a bidirectional connection means the access allows both writing and reading operations, while with a directed connection, reading is allowed if the shared resource is the source and writing is allowed when it is the destination.

SUBPROGRAM CALLS AND ACCESSES – As described before, a subprogram represent a unit of executable code, however a single unit or even a collection of them is not enough to describe the behaviour of a process; it is necessary to define the execution sequence. With AADL, it is possible to use different approaches to model the interaction between multiple subprograms, their local execution order and remote triggering. The main tool available is the call sequence defined in the `calls` section of a thread, inside

the sequence, identified by a name, the reference to the subprogram can be modelled in three ways. First option is to specify only the subprogram classifier, this approach can be used to just identify the subprogram and leave the actual local instance implicit. Alternatively, it is possible to define a binding between the called subprogram interface and the actual instance by using the property *Actual_Subprogram_Call*, this option is useful to model remote procedure calls or to define the implementation in a single location and then reference it in multiple places. Last option is to reference a subprogram access; this type of feature access works in the same way as a data or bus access, it provides or requires access and specify the category of the reference subprogram. The key difference is that the connection between two subprogram accesses is always bidirectional since the source is defined in the call sequence and an executable routine is expect to return to the caller.

FEATURE GROUPS – They represent a collection of component features or other feature groups. Feature group types, i. e., a set of component features, defines the internal structure of the group, they can be composed of any type of feature with any direction (i. e., any in and out port and any provides and requires access). Feature groups have multiple applications, for example can be used to simplify the model at higher level of details, or to model multiple communication channel always operating together, or to provide an abstract definition of a component to be refined later. On the component frontier, only the feature group is visible, and connection can be defined only between groups with the same type. Inside the component, the feature group acts as a reducing interface where all the compatible internal features converge.

4 MODELLING

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena.

— John von Neumann

As said by John von Neumann, science is about making models. They can be used to simplify a phenomena and make it easier to understand and define, moreover, a model can be used to quantify or visualise reality. Knowledge extracted by the process of modelling can be reused to create a simulation (i. e., another form of modelling) of the system under analysis. For all these reasons, and because is one of the most innate ability of humans, modelling has always been the cornerstone of science, engineering and arts.

Modelling in engineering is an essential tool for design, analysis and simulation, models have different characteristics and take various shapes. A collection of mathematical formulas can be used to describe a physical phenomena (e. g., friction between the ground and the wheels), or the behaviour of a system (e. g., a control system). Differently, a flow chart is a graphical model of an execution process, while pseudo-code is a textual one. A 3D model capture the physical shape of an object and can be used to study the design or the space occupancy.

This chapter presents various modelling techniques that we used to describe multiple aspect of the architecture of a robot. First, we introduce the *component-and-connector* paradigm, its basic concepts and why it is used in robotics. From there, we move to the AADL representation of the paradigm, and an extension to cover first a ROS node and later a complete architecture. To conclude the description on modelling robotic system, we introduce a series of

template to simplify the modelling process. Lastly, since AADL is not a data modelling language, we present two approaches based on Abstract Syntax Notation One (ASN.1) and JavaScript Object Notation (JSON) to model the data exchanged in the system (i. e., ROS messages) and the internal state of each component.

Contents

4.1 THE COMPONENT AND CONNECTOR PARADIGM

In robotics, the most popular middleware and frameworks are based on a *component-and-connector* paradigm, while different approaches implement it in different ways, the underlying conceptual structure is the same. In ROS, it is represented by the computation graph, a peer-to-peer network of processes managing and exchanging data together. Here, following the usual terminology of graphs, the components are called nodes, while the connections are represented by asynchronous topics or synchronous services; in both cases the communication happens by exchanging messages. In SmartSoft, the underlying technological approach of the SmartMDSD toolchain, the structure is based on components, communication patterns and communication objects. Components are interconnected to each other by using one of the four possible communication patterns: two synchronous, based on a client/server paradigm (i. e., send and query), and two asynchronous, based on a publisher/subscriber paradigm (i. e., push and event). All the patterns communicate by exchanging communication objects. Another example is the Robot Construction Kit (RoCK), which is based on the component model of the Orocos Real Time Toolkit (RTT). In RoCK, and consequentially in Orocos, the architecture is, again, based on components connected to each other through ports. One last example is the OpenRTM-aist middleware developed by the Japanese National Institute of Advanced Industrial Science and Technology. They fully embraced the component-based approach, where a robot is made by multiple subsystem, and each of them is a collection of components. Components communicate to each other using connections established between predefined ports.

The popularity of the *component-and-connector* paradigm is not coincidental. In their structure, robots are a system of systems, a hierarchical collection of components interconnected together to create a working apparatus. Physically, a robot is a collection of sensors and actuators; same goes for the behaviour, simple low-level independent functionalities are not enough to implement

even the simplest robot. Given all these needs, the most natural approach is to decompose the system in different and simpler subsystems and to simplify and characterise their interactions by the use of interfaces; the result is a *component-and-connector* paradigm.

The aim of this work is to provide a general and flexible representation that can be used to model an architecture that captures the design a robot and it is compatible, at least at an higher level of specification, with multiple middleware and frameworks. To do so, we defined some common design patterns often used when creating robotic architectures. As already mentioned, the first key design approach is the use of components and connections, but, of course, this is a very high-level description and it is only useful to define the topology. To better define an architecture without including technological details (i. e., specify a middleware or framework), we have to define a general description for component functionalities and we have to specify the nature of each connection. By analysing the existing solutions for robotics (i. e., ROS, SmartSoft and Orocos/RoCK), we identified four possible component behaviours that can exist (and coexist) in a component.

SOURCE – A component expresses a *source* behaviour when it is a generator of data or events. An example is a simple ROS node implementing a publisher, it generates messages and circulates them in the runtime graph. In SmartSoft there are two communication patterns that, implemented in a component, evoke this behaviour: push, to generate message and send them to other components, and event, a data-less communication to trigger action in other components. This type of behaviour is used for device drivers, since they create and circulate a digital version of the analogue input they detect, but also it used by coordinator components, they are in charge of initiating high level functionalities by generating an event or a specific message.

SINK – A *sink* is a component that consumes data or events. In ROS, a node is a sink when it implements a subscriber that receives, processes, and consumes messages. The counterpart in SmartSoft is, again, a component that implements the same two communication patterns (i. e., push and event), but this time it

is on the receiving side of messages and events. This type of behaviour is implemented by component controlling actuators, they receive commands from other components and consume them to operate a physical device. Alternatively, it is used by storage managers or loggers, they simply collect all possible messages and store them. Lastly, any component activated by an event implements a corresponding *sink* to receive and mange the trigger.

FILTER – The most common behaviour for a component is the *filter*. This type of component receives messages or events as an input and processes or relays them to the create an output. To describe more precisely the internal functioning of this behaviour, we have to distinguish two categories: without or with memory.

In the former, the component does not store in any way the data received, they are processed and directly re-circulated in the system. This approach is common when doing simple conversions (e. g., change the unit of measurement or the coordinate system) or when it is necessary to resample the data (e. g., change the frequency or zero-pad the messages). In ROS, it is implemented by processing the received message directly in the subscriber callback and publish it before the leaving the callback environment. The latter behaves similarly to a combination of a *sink* and a *source*; messages or events are received by the component and stored locally, then, at a later time, recalled from memory, processed and relayed in the system. This approach is used when doing more complex processing, for example when multiple messages need to be processed at the same time (e. g., smoothing a velocity set-point), or when multiple inputs need to converge in a single output (e. g., combining multiple laser rangefinder measurements in a single one). In ROS, this happens when messages are processed in a callback but not completely discarded at the end, and later, in the main loop or in a different callback, they are processed and circulated back in the graph.

REACTIVE – A component has a *reactive* behaviour when its functionalities are synchronously triggered by a message or an event, it is usually implemented by using a remote function call. In ROS this kind of behaviour is exemplified by services; they offer a public interface that can be called by external components

and react with a synchronous execution of a function that may return a value. SmartSoft implements a similar synchronous system, but differentiate between a one way communication with no answer (i. e., send and forget) and a two way communication with a specific response. This behaviour is used to delegate to a central component a specific functionality (e. g., centralised conversion system), to activate a remote functionality (e. g., component re-initialization), or to guarantee a timely answer to a request (e. g., soft real time functionalities).

Implicitly, by describing the possible behaviours of a component, we already outlined the nature of the messages exchanged: a connection can be data-based or event-based. In a data-based connection, messages with a specific content are exchanged between components. Usually, a specific communication channel only support a pre-defined data format, but, in theory, it is possible for a data-based connection to not specify the nature of the message exchanged. Each middleware or framework uses a different language to describe data format, for example both ROS and SmartSoft use their own domain specific language to define communication objects. This is source of a strong and unnecessary fragmentation, because in the end all middleware and frameworks rely on basic data types and data structures, that are already covered by numerous standard data description languages. Event-based connections fall on the opposite side of the spectrum, since there is no data exchanged, the key element is the communication itself. The receiver only needs to detect an active connection to collect the event generated by the sender. In practices, it is common to implement event-based communications as data-based communication carrying an empty message, this is the approach used by ROS, since it doesn't support any pure event-only communication, but any communication channel (i. e., topics and services) supports standard empty messages. In SmartSoft, event-based communication is supported only paired with a publish/subscribe paradigm, since the *Event* communication pattern exists only for asynchronous communications.

The complete the description of an architecture based on the *component-and-connector* paradigm, we need to analyse the car-

dinality of the connections. Potentially, there are four different cardinality:

- *1-1*, this is an exclusive connection where the a source is directly connected to exactly one destination.
- *1-n*, in this type of connection the stream of messages or events generated by a single source can be connected to one or multiple destinations.
- *n-1*, to achieve this configuration multiple sources need to converge in a single destination.
- *n-n*, this is the most permissive of all cardinalities. In this configuration there is no limit in the number of sources and destinations for a single communication channel. By using a local policy to this type of communication it is possible to implement all the previous ones, however when a communication is *n-n* by design it is impossible to guarantee other cardinalities. The result is a very flexible, yet unpredictable connection.

In practice, robotic middleware and frameworks, rarely implement all four options. In ROS, topics are implemented as pure *n-n* communication channels, since they follow a *publish-and-forget* paradigm. Any number of publishers can publish messages on a topic and each subscriber receive a copy of the message. There is no ownership of a messages after it is published on a topic. While the communication supports a full *n-n* cardinality, in practice, it is never used as such, but always as multiple publisher and a single subscriber (i. e., *n-1*) or a single publisher and multiple subscribers (i. e., *1-n*). ROS services are inherently *n-1*, since only a single server can provide a specific service, while multiple clients can request it. SmartSoft is more strict in the definition of the cardinality of its communication patterns: all the patterns with a publish/subscribe structure (i. e., push and event) have a *1-n* cardinality, all the patterns following a client/server approach (i. e., query and send) support only a *n-1* cardinality.

4.2 AADL FOR ROBOTICS

The Architecture Analysis & Design Language was originally developed in the field of avionics, then it was redesigned to target embedded real-time systems; therefore, never in its history the language was specifically designed for robotics. However, a lot of parallels exists between embedded and robotics systems, consequentially, characteristics that were designed for the former are more than suitable for the latter. Moreover, a general design approach means the language is not bound to a specific field and its design was not conditioned by the existing methodologies and technologies. Its agnostic nature makes AADL an excellent choice to provide a general modelling language for robotic systems, different middleware and frameworks sharing similar design principles can be represented easily with a common language. Additionally, AADL formal syntax and expressiveness guarantee consistency in the models and reduce the necessity to introduce extensions and ad hoc modifications, and even when this is necessary, they are regulated by the language itself.

As described in Section 3.2, AADL provides modelling tools for both hardware and software components. This is expected from a modelling language designed for embedded systems, where the development of the software components is tied to the hardware platform, however, this characteristic is extremely useful for robotics, too. Thanks to the abstraction provided by component-based middleware, modern robotic systems are not tightly connected to the underlying hardware platform as they used to be; today an obstacle avoidance system does not need to know exactly the data format of the measurements provided by the laser rangefinder to work correctly. However, this is true only for the development of single components or compartmentalised set of components, when designing the whole system or during execution, it is necessary to take in account the behaviour of both hardware and software. *How many and which sensors the robot uses to localise itself? Is there a teleoperation system? How much time it takes for a measurement to propagate in the system? How many critical functionalities are interrupted by a faulty sensor or actuator?* All these critical questions have to be answered during the design phase of the

system, and this can be done by correctly modelling the hardware (sensors, actuators, connections, execution platforms, etc.), the software (drivers, low-level interfaces, functionalities, etc.), and their interactions.

In AADL, a designer can use properties to specify the finer characteristics of each component. Some examples are memory size, processors computational power, processes resource consumption, or connection throughput. All these properties can be used to perform a formal analysis of the system before deployment, or even before the implementation. One of the tools provided by AADL is the concept of flow, they are logical path through the architecture and they are specified from a component input to a component output. These flows can go through any type of feature (i. e., ports, accesses, groups and abstracts) and can represent any logical pathway (e. g., data, control, fault event, etc.). When modelling a flow, it is necessary to specify the source, the sink and the complete path of the flow, however, the definition can be done at system-level, it is not necessary to specify the behaviour of the flow inside the components. From this specification, it is possible to do end-to-end analysis, for example, identify the component involved in a critical communication, estimate error propagation, calculate the time necessary for a measurement to impact on the behaviour of an actuator. Information and analysis about propagation time of messages and latency are fundamental in hard real-time application, but even soft real-time application can benefit from a strict performance analysis. For example, an high-speed delta robot need to consider communication latency to operate with high precision, or in an autonomous wheelchair, while the control system doesn't need high reactivity given the speed involved, a correct estimation of the latency will make the difference between a sudden braking and a gentle slowdown when faced with an unexpected obstacle.

On a more technological level, the detailed description of components, their interactions, their structure and hierarchy provided by AADL is extremely useful to solve some intrinsic problems of robotic middleware. For example, if we consider ROS, there is a total absence of an architectural view of the system. A partial representation of the interactions of the components is available at

runtime by using tools like `rqt_graph`, but this representation only consider topics and does not reflect the full structure of the system. Something similar can be achieved during deployment by using launch files, but while they are useful to organise the runtime of the components, there is no way to visualise the interactions between them and they do not capture the internal connections. Both this limitations can be solved by using AADL, a complete model of the architecture provides a view of the system since its inception and by using the *system* component is possible to recreate the same hierarchy provided by launch files. Even when considering middleware more focused on a model-based approach, AADL can bring great advantage. In the SmartMDSD toolchain, architectures are designed using a custom meta-model defined via the Eclipse Modelling Framework, this approach limits the design process to a very specific environment and tightly connects the design phase with the implementation phase, since the model reflect exactly some specific software artefacts. Moreover, while quite complete, the SmartMDSD meta-model, do not consider the hardware components as part of the architecture, therefore any analysis related to the hardware has to be done with additional tools. In this case, AADL could be used as a general high-level modelling language compatible with the SmartMDSD meta-model, to be then transformed in an intermediate representation compatible with the existing code generators. The AADL version of the model could be used to perform analysis and to speculate on the possibility of using different technologies to implement different components, or to deploy the same design on different platforms.

While not strictly related with robotics, an important feature of the language is the active community, and the available tools. The Open Source AADL Tool Environment (OSATE) is a development environment not only to develop AADL models both graphically and textually, but also to exploit all the validation and analysis capabilities of the language. Some examples are: end-to-end latency analysis, port connections consistency checks, computer resources budget analysis. Moreover, OSATE act as an interface for the capabilities provided by Ocarina, an AADL model processor that supports parsing, code generation and model checking. Ocarina

uses a front end/back end paradigm, where the front end is the AADL parser and a back end can be any code generator that goes from the intermediate representation provided by the front end to an executable code. Ocarina already support various targets for code generation, none of these is a robotic middleware, however, thanks to the front end/back end paradigm, we created a suitable code generator that goes from an AADL model to a complete ROS architecture. This code generation process will be described in details in Chapter 5.

MODELLING THE C/C PARADIGM IN AADL

AADL is the perfect candidate to describe architectures based on this paradigm, because the language itself, while centred around the concept of schedulability of threads, is designed to support the structure defined by components and connections. AADL components, at any level (e. g., system, process, device, subprogram, etc.) support, some form of feature (i. e., ports and accesses) to communicate and interact with other components. In the previous section we described all the elements useful to characterise the robotic components and their interactions (i. e., component behaviours, type and cardinality of connections), in this section we will describe how the same concept can be modelled by using AADL.

First of all, it is necessary to outline which AADL artefacts are necessary to model this kind of paradigm. As described in Section 3.2, the general top level container is the *system* category, one step below there are three possible software categories that could act as components: *process*, *data* and *subprogram*. A collection of subprograms as subcomponent of a system is used to model libraries or sets of APIs, data as first level subcomponents model general storage or shared resources, processes are used to describe an enclosed execution space and can exist only as direct subcomponents of a system. Therefore, it is clear that the perfect candidate to model components in the *component-and-connector* paradigm is the *process* category. Since the execution space of a process is directly accessible only by its own subcomponents, the

category supports, on its frontier, all available AADL features (i. e., interfaces).

We described in the previous section how communication can be data or event-based, AADL supports both this types of communication thanks to the additional type specifier when defining a feature, it can be a *data port*, an *event port* or an *event data port*. Moreover, ports can be characterised by a specific data type, defined using a *data* category, this is useful to model the type of message exchanged in the connection. In AADL, all ports have natively a *n-n* cardinality, with the exception of the *input data port*; since this type of port models a communication with no queue, it does not support multiple incoming connections, however they are supported by the *input event data port* since it includes a queue in its model. In the base language, there is no way of restricting the cardinality of ports, but it is possible to define extra properties to better specify a connection and, eventually, indicate its cardinality.

To model component behaviours, it is necessary to go a step further in the component hierarchy and to take in account the potential subcomponents of a process. Again, the *data* and *subprogram* categories are potential subcomponents, but are not relevant to describe the behaviour of a component, since they are used to define memory and subroutines. The third, and more suitable, candidate is the *thread* category. They represent an execution path through code and their execution behaviour is periodic with various characteristics or triggered by an external input (e. g., incoming data or events), moreover, multiple threads can coexist in the same process and, potentially, execute in parallel. All this characteristics make the thread a suitable category to model component behaviours. However, as we described in the previous section, each behaviour is related to a specific interaction outside the frontier of the component, therefore to completely characterise it, it is necessary to combine a thread with a specific port.

SOURCE – In this behaviour the components generates messages or events. Therefore the thread needs an output port that can be of type data or event. Since there is no external source triggering this

behaviour, it is necessary to specify the scheduling of the thread, usually periodic.

SINK – In this case the components receives and consumes messages or events. Given this, an input port needs to be modelled as interface of the thread. By changing the type of the input port it is possible to change the how the sink behave. A data port, combined with a periodic scheduling, would model a sampler that receive a constant stream of data, an event data port would define the subscriber of a publish/subscribe paradigm, an event port defines a sink managing external triggers.

FILTER WITH MEMORY – For this type of behaviour it is necessary to add the extra memory element, in AADL all memory, on a software level, is modelled using the *data* category. Therefore, to completely model this type of filter, it is necessary to define a data component inside the process that acts as a shared memory between two threads, one with a input port (i. e., sink equivalent) and one with an output port (i. e., source equivalent).

FILTER WITHOUT MEMORY – This type of filter is easier to model, since there are less elements. All processing happens directly in the thread that receive the message and the same thread is responsible to circulate it back in the architecture. Given this, this behaviour is modelled using a single thread with an input port of any kind and a corresponding output port.

REACTIVE – This behaviour represents a synchronous remote execution similar to a remote function call. In AADL there are various approaches to model a RFC, to keep the structure more in line with the other behaviours the best option is to pair a thread with a remote subprogram call. To do so, it is necessary to define a *subprogram* as a subcomponent of a thread and, on the frontier, declare an access bound to that specific subprogram.

A BASIC EXAMPLE

With the paradigm described in Section 4.1 and the tool provided by AADL presented in Section 4.2.1, it is possible to define a basic

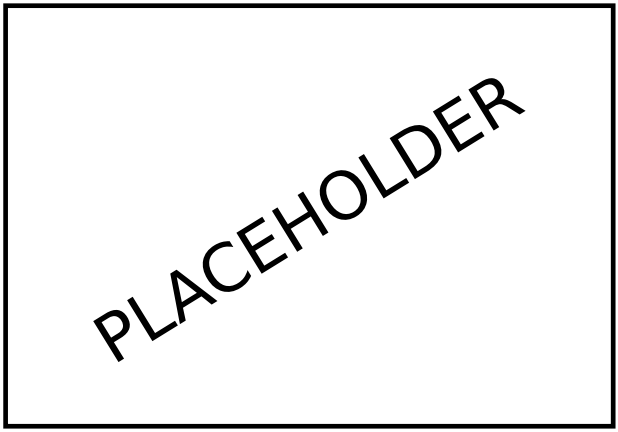


Figure 4.1: TODO

architecture, not only the topology of the connections, but also the internal functioning of the components. As an example, let us take the architecture defined in Figure 4.1; in this simple architecture we want to model a robot with two control functionalities: line following and teleoperation.

In the top branch of the architecture we have the line following subsystem: a sensor driver that generates measurements indicating the presence of a black line on the ground and a simple component that directly translate the sensor input in a velocity command. The former expresses a *source* behaviour, it generates messages and circulates them in the system, it is modelled by a process containing a periodic thread, on the frontier there is a output data port. The latter has a *filter* behaviour, it has two ports on its frontier: one input event data port, to queue the messages and trigger a functionality of the component, and one output data port, to send to the next component the velocity commands. In this example, the line following component is very simple and can estimate a set-point directly from the sensor measurements, this means that it does not need to store any information: it is a filter without memory. The internal model of the components is a single thread with an input and an output port, both directly connected to their corresponding ports on the process frontier.

On the bottom branch there is the teleoperation subsystem: a joypad driver providing readings of the input and a teleoperation component to convert the joypad input in set-points. Similarly as the other branch, the driver expresses a *source* behaviour, it converts the input coming from the joypad in messages compatible with the architecture. The teleoperation component, however, does not have the same *filter* behaviour of the line follower, since it stores the incoming messages to smooth the set-point: it is a filter with memory. To model it, it is necessary to define on the frontier of the process two ports: one input event data port and one output data port. Internally, these two ports are connected to two different threads, one is triggered by the event data port, it receive the messages and store them locally. The output data port is connected to a periodic thread, on fixed intervals it reads the two most recent messages and estimate a single set-point compatible with the robot acceleration. Each thread has its own data access to write or read on the shared memory area, modelled as a data component.

In the middle of the architecture, to mediate between the two different control systems, there is a multiplexer component. This is an example of multiple behaviours coexisting in the same component. There are two filters, one for each input, their functionality is to relay the correct message to the output and only one of them is active at a given time, additionally there a *reactive* behaviour. One of the two inputs (i. e., line following or teleoperation) can be selected by triggering a remote function call on the component, which changes a local variable specifying the active input. To model this component it is necessary to define four different features on the frontier: two input event data port, they are the receiving side of the two filters, one output data port, it selectively relays the selected input, and one subprogram access, to trigger the reactive behaviour. Internally, each input port is connected to an independent thread, triggered by the port itself, this threads receive the inputs and store them in a shared memory area. The output port is connected to a periodic thread, which reads the content of the inputs from the shared memory and publish the correct one. The subprogram access interfaces with a thread which has the corresponding subprogram as a subcomponent, this thread is

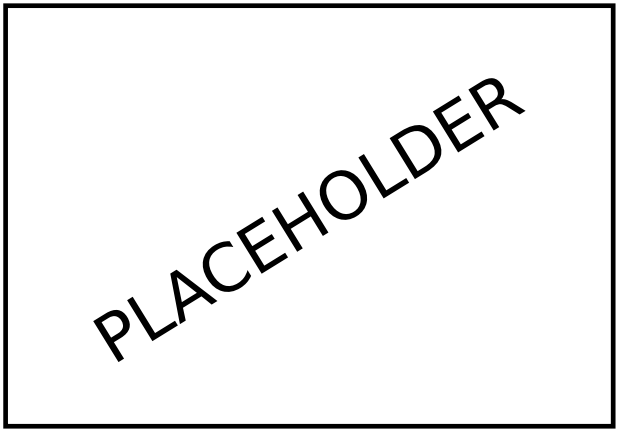


Figure 4.2: TODO

triggered by an external call on the access and changes the current selected input. All the threads have a data access to the same data component representing the shared memory area.

The last element of the architecture is the control component. It is directly connected to the hardware of the robot and consume the set-point provided by the multiplexer to operate the motors, in summary, it expresses a *sink* behaviour. It is modelled by a process containing a single thread, which is connected and activated by the event data port defined on the frontier of the component.

4.3 FROM C/C TO ROS

In Section 4.1 we already highlighted parallels between the *component-and-connector* paradigm and ROS. At first, it may seem the architectural model of ROS is more complex than a simple component-based approach, since, together with the nodes, there is the extra element of topics, an apparently independent and additional factor. However, topics are nothing more than an agreed name to establish a connection between nodes, they are defined and treated as existing entities in the runtime graph, but they do not

mediate any communication. After two nodes agree, through mediation of the ROS master, on a communication channel (i. e., a topic), any consequential exchange of messages happens on a direct connection between the two nodes. This means a topics is an aggregation of point-to-point connections and not a independent component mediating the communications.

Given the actual nature of nodes and topics, it is possible to say that ROS completely follows a *component-and-connector* paradigm, where the nodes are the components and the topics are an aggregation of one or multiple connectors sharing the same characteristics. Figure 4.2 illustrates how all the possible topic configurations (i. e., *1-1*, *n-1*, *1-n* and *n-n*) can be translated in an AADL-based representation, assuming that all the output ports are data ports and all input ports are event data port. The reason of this choice is to model the behaviour of ROS publishers and subscribers. The former does not produce an event notification when it creates a message, it just circulates them in the graph, therefore it is a simple data port, the latter is triggered by the arrival of new messages and supports queue, for this reasons it needs to be modelled as an event data port. As shown in the figure, the substitution process from a topic representation to a connection-based one is quite straightforward. First we select a topic, then we list all the publisher interacting with it, after removing the topic, we directly connect each publisher to all subscribers of the original topic.

However, by doing this substitution process one piece of fundamental information is lost. In AADL, each connection and each port requires an unique name, this means that it is impossible to maintain the information that a specific connection is part of the aggregation defining the topic. An easy solution is to give the connections a recognisable name, for example each connection for the topic `/chatter` can have a name starting with `chatter_`, however, easy does not translate to good, since this approach is completely unacceptable, it breaks the model by encoding information in variable names. A more elegant solution compatible with the features of AADL is to declare a new *property set*, in this set, called `topic_properties`, it is possible to defined two new properties for connections and features. One property is the *Default_name*, it applies to ports and subprogram accesses, it is used to specify the

name of the topic (or service, for subprogram accesses) defined during the implementation of publishers and subscribers. The other property is *Name*, it applies to connections and, at system level, it can be used characterise connections between processes as part of the same topic. The utility of this property is twofold, not only it capture the information of a connection belonging to a topic, but it can be used to include in the model dynamic renaming of topics typical of ROS.

When introducing the concept of component behaviours, we showed how each behaviour can be represented with a specific ROS functionality, however this does not mean that the relationship is bidirectional. ROS imposes very few restriction to the developer, therefore, while the external interface of a node follows the same behaviours presented in Section 4.1, the internal implementation may follow an arbitrary pattern. The basic implementation style of a ROS node is to have a main execution loop that periodically polls subscribers and services and pass them the execution every time a new request arrives, similarly to the *component-and-connector* paradigm we presented, this execution happens in a separate environment, however, it is not independent and parallel, but sequential. Practically, this difference it is not meaningful, the conceptual structure presented by the model is the same and this is just an implementation detail. Nevertheless, it is worth to explore an implementation where the actual execution matches completely the description of the model; this will be discussed more in details in Section 5.2. In summary, *sink* and *reactive* behaviours are related to their counterpart in ROS by a bidirectional relationship. It is not possible to say the same when we consider *source* and *filter* behaviours. For reference, let us take the simplest ROS node with publisher functionalities presented in one of the basic ROS tutorial[1]. In this example, it is possible to see how it is common practice to define a single execution path where everything (e. g., initialisation, polling, publishing, etc.) happens, while this approach is sustainable for small implementations and node with simple behaviours, it is not suitable for more complex components. This is especially true when implementing compon-

1 http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)

ents that express multiple *filter* with memory behaviours, in this cases to ensure maintainability, flexibility and understandability of the design it is necessary to follow an approach more in line with the *component-and-connector* paradigm, where each different input and output is managed by an independent execution path. In summary, given the flexibility of the ROS middleware, it is not possible to guarantee that all the implemented node will follow, internally, the design of the *component-and-connector* paradigm. However, the aim of this work is to provide a general modelling approach that can be used to enhance the existing design practices for robotic software, therefore, we will present a way to model (see Section 4.3.1) and implement (see Section 5.2) ROS nodes following a simpler, more flexible, easier to maintain and more robust design aligned with the *component-and-connector* paradigm.

While describing the relationship between component behaviours and ROS node implementations, we introduced a key element of any robotic component: the main execution loop. It is in charge of multiple critical tasks of the component; some examples are initialisation, management of incoming requests, dynamic reconfiguration (when supported), error management, shutdown and clean-up procedures. Its role is very important but it is often left out in the modelling phase because of its implicit nature (often hidden by the framework) as the backbone of the component. Nevertheless, it is very important to include it in the modelling process, because some of its tasks may need to be specialised by the developer (e. g., initialisation, error management, etc.). In ROS, in particular, given the freedom left to the developer, it is critical to model the main execution loop to highlight the different functionalities of the nodes, since, as described before, they are often merged in a single execution path. This is another case where for the sake of creating a more general approach and to enhance the ROS design of nodes we, with our modelling approach, overrode the flexibility of ROS to make the main loop an independent execution path disconnected for any other functionality-related path. In summary, in our model, each ROS node, independently from its behaviour, has an additional thread that manages the core functionalities of the component.

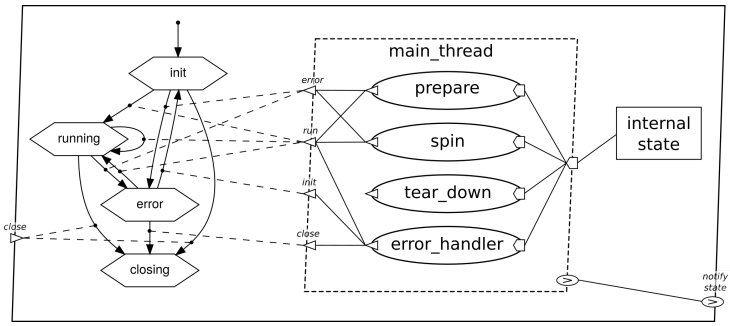


Figure 4.3: Graphical representation of the AADL description modelling the base structure of the enhanced ROS component.

MODELLING A ROS NODE IN AADL

In the previous sections, we introduced all the elements we need to model a ROS node: the *component-and-connector* paradigm, how it applies to ROS and the missing elements to bridge from a conceptual model to technological design, and the corresponding AADL description. In this section, we provide a full description of a ROS node, providing first a model of an essential (i. e., no component behaviours) component and then modelling additional functionalities.

Figure 4.3 provides a graphical representation of an essential ROS node. At first glance, it is clear that the model captures more information than what we described until now: the component contain a state machine that is triggered by multiple event ports, the main execution path is detailed by various internal functionalities, there is a permanent internal state and two new features appears on the frontier of the component. Let us analyse the model in order by starting from the state machine: it is used to model the internal life cycle of the component. In ROS there is no defined evolution of the status of a node, usually it can be in only two different states: not executing and active, and the only way to check the current status is to trigger one of the functionalities of the node (e. g., read a topic). However, it is common in component-based approaches to define a clear life cycle of the

component, moreover, other robotic middleware and frameworks have an established evolution of the status of their components (e. g., SmartSoft and Orocos), therefore, modelling the internal state machine is a requirement to guarantee the generality of our approach. Our life cycle follows a quite standard structure with four different states.

INIT – Initial state of the the life cycle, all the initialisation procedures happen here. After leaving this state it is guarantee the component is ready to execute all its functionalities. From *init* the component can transition to *running*, when all the initialisation procedure are completed successfully, *error*, or *closing*.

RUNNING – Normal operational state of the component. In this state all the main functionalities are active and the component is working with no issues. The possible transition are: to *running*, the state machine periodically triggers a self loop to ensure that the component is alive and functional, to *error*, or to *closing*.

ERROR – The component transitions in this state when it captures known errors. While in this state the component is not in execution and recovery procedure can be activate to correct the errors and transition back to an active state (i. e., *init* and *running*), or if the error is unmanageable or catastrophic, to transition to the *closing* state.

CLOSING – Final state of the life cycle, all the clean up and shut down procedure happens in this state. The transition to this state is triggered by any shutdown signal (internal or external), or when the nodes encounters a catastrophic error that cannot be recovered and requires the node to shut down.

Transitions in the state machine are triggered by event ports on the main execution path of the component and on the external frontier. The event port on the frontier is used to model any external signal used to force the shutdown of the node (e. g., SIGINT signal), it triggers the transition to the *closing* state from all active states. All the ports on the main execution thread represents the normal evolution of the system and are activated every time

one of the execution phases of the component finishes or when it encounters an error.

The main execution path of the component is modelled using a periodic thread called *main_thread*. The thread has four subprograms as subcomponents and they represent the active functionalities during each phase of the life cycle. This relationship can be enforced by AADL, since for each state it is possible to specify exactly which subcomponent is active, therefore, while the *main_thread* is always enabled, only one of its subprograms is active at any given time. Since each subprogram is in charge of a specific state, they are all connected to their corresponding event port to trigger a transition to the next state. Each subprogram has a specific functionality.

PREPARE – It is active in the *init* state. This subprogram is in charge of managing node initialisation; it sets parameters, initialise variables, set up publishers and subscribers, and any other node specific initialisation activity. It is connected to the *run* and *error* event ports, to trigger the two possible transitions related with a successful or unsuccessful initialisation.

SPIN – It is active in the *running* state, and here the ROS spinner is implemented. When this subprogram is active the *main_thread* acts as a coordinator of node behaviours, it checks for incoming events and handles potential errors. Two event ports are controlled by this subprograms: *run*, to trigger the self-transition and keep the component active, *error*, to transition to the *error* state when necessary.

ERROR_HANDLER – It is active in the *error* state. Here are implemented all the functionalities to deal with known errors of the component, for example an incorrect initialisation or a malformed message. From the *error* state it is possible to transition to any other state, as a result this subprogram is connected to the *run*, *error*, and *init* ports.

TEAR_DOWN – It is active in the *closing* state. This subprogram implements all the procedures related to node shut down. For example, in some cases a node may need to notify another before

shutting down, or it need to gracefully disconnect from a physical device. Since this is the final state of the life cycle, this subprogram is connected to no event port on the thread frontier, even so the corresponding port on the subprogram is still modelled for flexibility and potential future extensions.

As a final note on the node life cycle, we have included in the model of this essential node a *requires subprogram access* on the frontier of the *main_thread* which is connected to its counterpart on the process. This access models a remote function call to notify an external supervisor about the current state of the node and any transition. In this way, not only the component has a clear life cycle that defines the execution phases, but it is possible to monitor the evolution of the status through time and keep track of any unexpected behaviour.

Last element to cover is the *internal_state*, it is a data component used to model the execution memory of the component. In Section 4.1, we described how not all the component behaviours requires a shared memory area, therefore, it should not be necessary to include a data component in an essential node. However, in practice, only the simplest component does not require a persistent internal state, because this shared memory area does not exist only as a way for different thread to exchange information. It stores configuration parameters, error mappings, information for shutdown procedures, etc., for this reasons, not only it is present in this minimal node, but it directly accessible by all subprograms in the *main_thread*. Since AADL is not a data modelling language (see Section 3.2), we will not detail here the inner structure of the *internal_state*, this topic will be covered in Section 4.5.

After modelling an essential node, and by using it as a starting point, we can now add behaviours and functionalities. Figure 4.4 shows a graphical representation of a more complex node expressing two coexisting behaviours: a *filter* with memory, modelled with a combination of a *callback* and a *publisher*, and a *filter* with no memory, defined with a subscriber and a publisher in the same thread and identified by the name *callback_pub*. As introduced in Section 4.3, behaviours are modelled using a combination of thread and corresponding ports. In the figure it possible to see

how each thread is characterised by a specific port that evokes its functionality: the *callback* has an input event data port, to receive messages and manage queues, the *publisher* has an output data port, to circulate messages to the rest of the architecture, and the *callback_pub* is a combination of both and has input and output ports. Each port on the thread frontier is then connected to its counterpart on the process to relay from the inside of the component environment to the outside, or vice versa. What is not visible from the figure is the data components associated to each port. In AADL, it is possible to specify the data type exchanged on a connection, and the model will automatically verify that the ports are compatible.

Listing 4.1: TODO

```
package std_msgs
public
    data String
        properties
            Source_Text => ("String.schema.json");
    end String;
end std_msgs;

package geometry_msgs
public
    data Pose
        properties
            Source_Text => ("Pose.asn");
    end Pose;
end geometry_msgs;
```

Listing 4.2: TODO

```
package publisher_subscriber
public
    with std_msgs, geometry_msgs;

    process complex_node
        features
            callback_in: in event data port std_msgs::String;
            publisher_out: out data port std_msgs::String;
            callback_pub_in: in event data port geometry_msgs::Pose;
            callback_pub_out: out data port geometry_msgs::Pose;
            close: in event port;
```

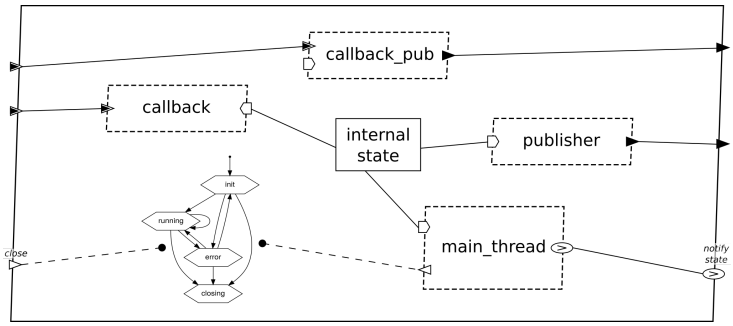


Figure 4.4: Graphical representation of the AADL description modelling a complete ROS node. The design includes two subscribers and two publishers.

```
        notify_state: requires subprogram access state_manager;
    end complex_node;

end publisher_subscriber;
```

Listing 4.1 shows a fragment of AADL code where data components associated to ROS messages are declared. Listing 4.2 shows how the same messages are included and used to specify the data type of the ports of the node presented in Figure 4.4. More details about ROS messages and their definition in the model will be discussed in Section 4.5.

MODELLING A ROS ARCHITECTURE IN AADL

In Section 4.2, we presented multiple reasons why AADL is a suitable language for robotics, one of them was the capability of the language to model both software and hardware components, however, in our descriptions on how to model ROS nodes we never mentioned any physical interface. The reason for this is that we followed a top down approach to describe a robotic architecture, at higher level of abstraction (i. e., in the *component-and-connector* paradigm) there is no need to make a distinction between a software component and an hardware component, but now that we are moving closer to the actual implementation, this character-

istics of AADL will be integrated in our model. We will specify how hardware components can be integrated in the architecture and how to differentiate between communications happening on ROS topic and other physical communication channels. When describing models closer to the implementation level, we also encounter design solutions that are not captured at a more abstract and general level, in ROS there are three key design features that we decided to specifically model in our approach. First of all, existing packages and nodes, the greatest resource of ROS is its repository of already available components, it is imperative to be able to model them correctly in an architecture. Second is *tf*, the coordinates frame manager, it is a backbone of various ROS nodes, therefore it is necessary to include it in the model. Last, it the *actionlib*, an interface to start, monitor, pre-empt and cancel remote tasks, while it is not a commonly used design approach, it is a very useful and elegant way to delegate and coordinate complex task between components.

PHYSICAL DEVICES – AADL offers multiple hardware categories to model the physical aspects of a system. For elements like processors or memories we will not go in details, since they work for robotic architectures in the same way of any other, it is possible to bound a software component (e. g., processes or data) to its physical counterpart (e. g., processor and memory) to specify the hardware implementation of the system. In ROS this feature of AADL can be used to model distributed architectures, by binding components to different physical platforms, the designer can specify where each node will be executed at runtime. This creates a deployment view of the system without the need of defining a different model, moreover, AADL analysis capabilities can be used to determine if a specific platform is suitable for a specific subset of the components (e. g., does the target computer has enough RAM to run the system?).

What is different in a robotic system is that sensors and actuators are integral part of it, to model them it is possible to use AADL devices. A *device* represents an interface between the physical world and the architecture, it can be modelled as a simple interface or to include the inner functionalities and characteristics

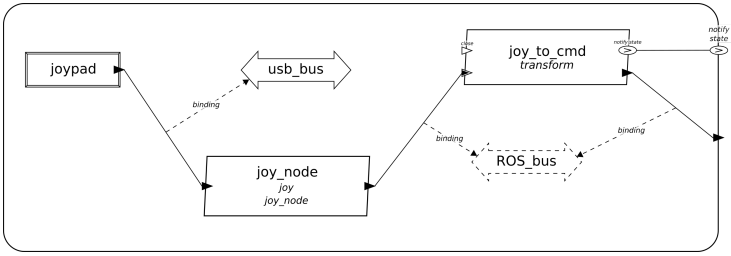


Figure 4.5: Graphical representation of the AADL description modelling a ROS-based teleoperation subsystem.

of the physical component (e. g., type of communication, computational power, data type). Devices can connect to processes using ports or accesses, in the same way as processes connect to each other. When modelling a ROS architecture, the fact that physical devices and software components communicate using the same interface rises the issue of differentiate between a topic-based connection and other types of connections; the solution comes in the shape of AADL physical and virtual buses. A *virtual bus* can be used to model abstract communication channels, like ROS topics, while a *bus* can be used for physical connections, like Ethernet and USB. Figure 4.5 shows how this two categories, together with a device, can be used to model a simple joypad-based teleoperation subsystem where a physical bus called *usb_bus* is bound to the physical connection between the device modelling the joypad and its driver, and a virtual bus called *ROS_bus* is bound to all the connections representing topics.

EXISTING ROS NODES – ROS is currently the most popular and widespread robotic middleware, this creates a very prolific and active community, which becomes one of its greatest resources. Given ROS component-based structure and popularity, a multitude of already existing packages and nodes exist that a developer can simply download and include in his architecture. When creating a modelling approach for ROS it is mandatory to include the possibility to model existing node, to do so we can exploit the dual representation based on component type and implementation provided by AADL. Existing ROS packages are modelled

directly as AADL packages, while existing nodes are modelled using the component type only, basically we provide an interface that appears and behave in the same way as the already existing component, but we do not model in any way the internal functioning. In Figure 4.5, *joy_to_cmd* is a custom made node, completely modelled as visible from the *close* port and the *notify_state* subprogram access, on the contrary, *joy_node* is an existing node from the package *joy*, here only topic related ports are modelled to provide an interface to the rest of the system.

Listing 4.3: TODO

```
-- Auto-generated process interfaces for amcl
-- Generated on: 17/11/2017 17:28:07
package amcl
public
with topic_properties, geometry_msgs, sensor_msgs, nav_msgs;
process amcl
    features
    amcl_pose: out data port geometry_msgs::PoseWithCovarianceStamped;
    particlecloud: out data port geometry_msgs::PoseArray;
    scan: in event data port sensor_msgs::LaserScan;
    map: in event data port nav_msgs::OccupancyGrid;
    initialpose: in event data port geometry_msgs::PoseWithCovarianceStamped;
    properties
    topic_properties::Default_Name => "/amcl_pose" applies to amcl_pose;
    topic_properties::Default_Name => "/particlecloud" applies to particlecloud;
    topic_properties::Default_Name => "/scan" applies to scan;
    topic_properties::Default_Name => "/map" applies to map;
    topic_properties::Default_Name => "/initialpose" applies to initialpose;
end amcl;
end amcl;
```

Listing 4.3 shows the textual AADL used to model the legacy *amcl* ROS package and the interface of the *amcl* node. It is possible to see how the model descriptions follow the same naming conventions of the original package. The ROS package name correspond to the AADL package, the process name to the node, and the topics name match the definition of ports and properties.

The only potential issue is to create all the models for the existing components, there are few possible solutions that combined can almost automatically generate them. First, analyse the existing nodes at runtime to detect which topics and which messages they

use, additionally it is possible to do code inspection to list all the publisher and subscribers, lastly, most packages are documented on the ROS wiki[2] with, at least, the list and type of topics. Unfortunately, sometimes this is not enough, the joypad driver is one of those example, to detect automatically the connection with the physical device is extremely difficult, at this point the only solution is the human intervention.

TF – This package is one of the core functionalities of ROS and it exists to help manage multiple coordinate frames and transformations over time. Differently from other ROS features, from the point of view of the developer the access to *tf* does not go through any established communication channel (i. e., topics or services), but by using a set of APIs that directly access the distributed coordinate system, additionally, there is no need to start a node to enable it. Therefore, disregarding how *tf* is practically implemented in the system, we can describe it as a centralised resource where all the coordinate frames of the robot and their evolution in time is stored, and it is possible to read or update the content of this shared resource by using specialised APIs. With these assumptions, the best way to model *tf* is to use a single data component at system level that all the nodes can access through data accesses when necessary. The single data component stores the description of the coordinate frames and their evolution in time, while the data accesses represent the bidirectional APIs.

ACTIONLIB – Actions are an extension of ROS services, created to manage requests that are too computationally heavy for a traditional client/server approach. An action client can trigger a remote execution on an action server, then resume normal functioning while waiting for a response, if needed. While an action is under execution it sends periodic updates of its status and the original caller can terminate it before it finishes. The actual implementation of ROS actions is based on topics that are used to trigger or cancel the execution, provide the result, and get updates on the status. When implementing the action client and server in a ROS node, a developer has to use two already provided classes:

2 http://wiki.ros.org/joy

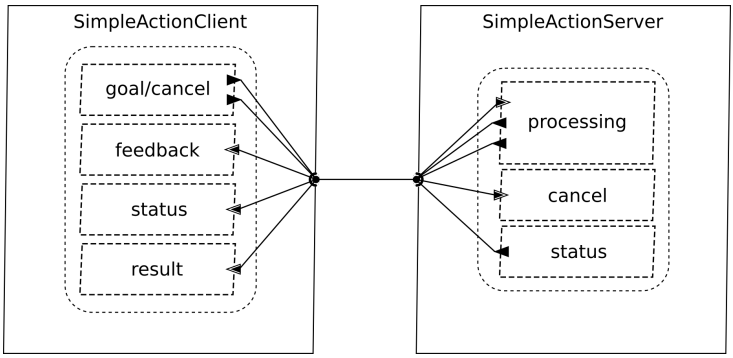


Figure 4.6: TODO

SimpleActionServer and *SimpleActionClient*. These two classes act as an interface hiding the underlying topic-based system. In AADL it is possible to do the same using a *thread group*; as the name suggest, it is a subcomponent used to group threads together and organize them. Fig. 4.6 shows a graphical representation of how an action can be modelled. The client thread group has two outbound ports representing the *goal* topic, used to activate the action, and the *cancel* topic, used to cancel the action; these two ports have their corresponding version on the server group as inbound ports, this time used to trigger the callbacks. The group modelling the *SimpleActionServer* has three outbound ports used to communicate with the client, these ports have their equivalent on the client group. Modelling is about abstracting the underling implementation and representing concepts, therefore at process level the ports of the thread group are aggregated in a *port group*; this maintain the conceptual representation of the action acting as a single communication channel.

A ROS BASIC EXAMPLE

In Section 4.2.2, we presented how to model a simple architecture using AADL and following the *component-and-connector* paradigm. In this section, we describe how that architecture can be updated to represent a complete ROS-based system. As visible for Fig-

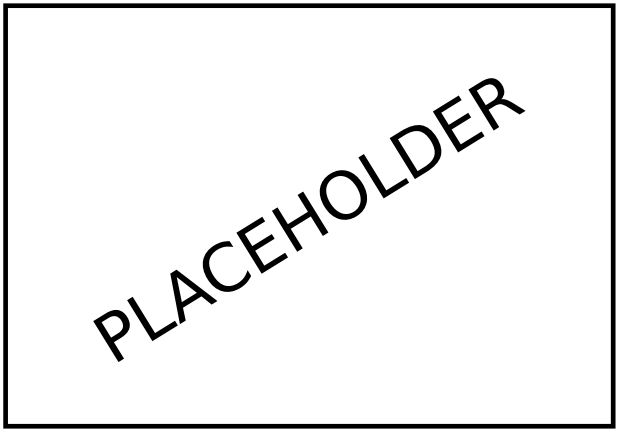


Figure 4.7: TODO

ure 4.7, the functionalities of the architecture are the same: line following and teleoperation. However, in this updated version, we include ROS specific elements and we model physical components of the system.

Let us start again from the top branch of the architecture: the line following subsystem. From a pure software point of view, the architecture is unchanged, it consists in two components, one expressing a *source* behaviour and the other a *filter* behaviour with no memory. The difference is in the internal representation of the components, now nodes, which is based on the essential node defined in Section 4.3.1; it includes the main execution loop, the internal life cycle and the internal state. Both these nodes are assumed to have application specific functionalities, therefore are fully modelled as custom nodes. From an hardware point of view, this subsystem now includes a physical device: the line detection sensor. It is modelled using an AADL device and it has a physical connection with the driver component. The connection is modelled using an output data port on the device and the corresponding input data port on the process, to specify that it does not represent a ROS topic, it is bound to a physical bus that models an USB connection.

The bottom branch is similar to the example presented in Figure 4.5. On the software side, the teleoperation component is unchanged, but now it is modelled as a ROS node instead of a generic *filter* with memory component, the driver component is replaced by the already existing *joy_node*, therefore it is modelled only using the external interface. As for the line following subsystem, now the teleoperation subsystem includes the physical device; it models an USB joypad and it is connected to the driver through a data port. Since this connection is a physical UBS connection, too, it is bound to the same physical bus as the connection between the line detector and its driver.

The multiplexer component is now a ROS node. The original *filter* behaviour is now replaced by a set of two subscribers, they collect the messages coming from the two different subsystems, and one publisher, it relays the correct message to the control component. The *reactive* behaviour is implemented using a ROS service, its functionality is the same, an external client can call this service to change the selected input from line following to teleoperation and vice versa.

The control component is now a control subsystem. In the previous version of the architecture, there was a single component expressing a *sink* behaviour, now, the component is replaced by a custom ROS node with a subscriber to receive the set-points and a device modelling the electrical motor. In this case the interaction between the driver and the motor goes through an Ethernet connection, therefore we modelled a second bus component representing this type of connection.

The architecture includes a virtual bus representing ROS topics, all virtual connections (i. e., topics and services) are bound to this bus. While not visible in the figure, the architecture includes data components for message types. Each type is represented by a data component in same package as defined by the existing ROS hierarchy. The data type of a port is specified in the process definition and port connected together must have the same data type.

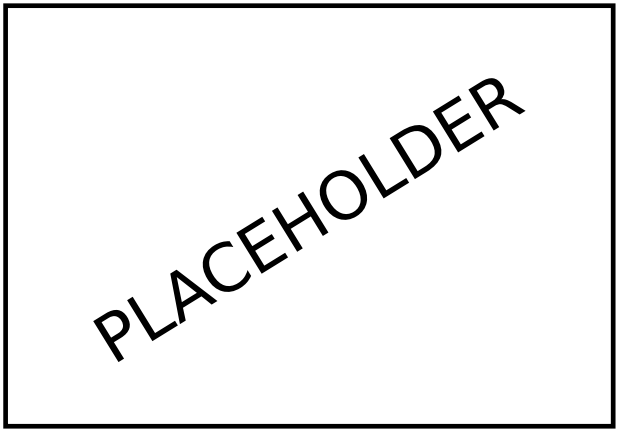


Figure 4.8: TODO

4.4 MODELLING TEMPLATES

With the model definition presented in the previous sections and the ROS-specific models introduced in Section 4.3.2, a designer has all the necessary tools to model a functional complete ROS architecture. However, this result does not come without complications, assuming a person is already familiar with AADL, the process of modelling a single architecture is quite tedious and prone to errors, especially replicate the same basic design for all ROS nodes. Moreover, this model aims to be a stepping stone for automatic code generation, for this reason it is necessary the structure of the model is consistent. In summary, there are few issues that need to be addressed: a) make the modelling approach more accessible, so it can be a reasonable alternative to the current development process for ROS, b) enforce the presented structure, without it being a burden to the design process, c) make the underlying structure self-explanatory, without the need of a lengthy accompanied documentation.

Solving completely all these issue is a very difficult task. In small size project, the process of modelling will always be more complex and time consuming than direct development, until we achieve perfect code generators. Moreover, there is a limit on

how much it is possible to enforce a specific structure, because there is a trade off between the flexibility of an approach and the number of functionality that is possible to model or implement. Nevertheless, in an effort to solve these issues, we developed a series of modelling templates that a designer can use to simplify the modelling process, to do so, we exploited the inheritance capabilities of AADL.

Figure 4.8 shows the hierarchy of packages and components when modelling a ROS architecture, and the frontier between the existing templates and the user-created model. At the top of the hierarchy there is the *component-and-connector* paradigm. Here, the main elements of the paradigm are modelled: the component and its implementation, the shared memory area and the inner functionality of behaviours.To model the threads of the behaviours, we used again a hierarchical approach: all behaviour interfaces inherit from the same parent thread which implements the access to the shared memory area as the only common characteristic, then they specify independently their own port to characterise the type of behaviour, only the implementation of the common parent exists since the internal description is always the same (i. e., a single subprogram).

The ROS AADL package extends the *component-and-connector* packages. Here all the necessary elements to model ROS nodes and architectures are defined. The ROS specific elements are modelled directly with no inheritance required, these are: the data component representing *tf*, the virtual bus bound to ROS communications, the main execution loop of the node, and the subprogram associated with the life cycle notification. All the other elements are inherited from the description of the *component-and-connector* package: the node and its implementation extend the component definition, each thread interface extends the corresponding high level behaviour interface (e. g., *service_provider* extends *reactive*, *callback* extends *sink*, etc.) while the thread implementation extends directly the behaviour implementation. One exception is the ROS timer, since it does not have a corresponding behaviour (it does not interact with the external environment) but it can be used to describe internal functionalities of the node or to give flexibility to the designer, it inherits directly from the

general *component_behaviour* thread. This create the equivalent of a "ROS node behaviour library". One element that is necessary to enforce when developing a ROS node is the data type associated with a specific topic, unfortunately the data type of a port is not a mandatory property in AADL. Our solution is to use AADL prototypes; they can be used to create a placeholder for any component or subcomponent. In the thread definition for ROS behaviours, we specified data prototypes associated with the input or output ports, the designer is then forced to specialise them before instantiating the model.

Listing 4.4: TODO

```
thread callback extends cnc::message_sink
    prototypes
        message: data;
    features
        msg: refined to in event data port message;
        tf: requires data access tf;
end callback;
```

Listing 4.5: TODO

```
process implementation complex_node.impl extends ros::node.impl
subcomponents
    callback: thread ros::callback.impl (message => data std_msgs::String);
    publisher: thread ros::publisher.impl (message => data std_msgs::String);
    callback_pub: thread ros::call_pub.impl
        (message_in => data geometry_msgs::Pose,
        message_out => data geometry_msgs::Pose);
connections
    pub_out: port publisher.msg -> publisher_out;
    cb_in: port callback_in -> callback.msg;
    cb_pub_out: port callback_pub.msg_out -> callback_pub_out;
    cb_pub_in: port callback_pub_in -> callback_pub.msg_in;
end listener.impl;
```

Listing 4.4 shows how prototypes are defined for the *callback* thread, a similar approach is used for all the other pre-defined threads. Listing 4.5 shows how they can be used to define the implementation of the node shown in Figure 4.4 and Listings 4.1.

At the lower level of the inheritance there is the specific package to be designed. Other than including the ROS AADL package,

it includes all the packages associated with existing ROS nodes and messages. When creating a new node the designer can extend the existing base node provided by the ROS packages, and then add all the necessary functionalities directly from the ROS node behaviour library. This makes the process of modelling a node follow the defined structure, while at the same time, it creates a more compact and easier to design model.

4.5 DATA MODELLING

So far, in this chapter, we described in details how it is possible to model component-based architectures together with their interactions, and how to transition from them to fully modelled ROS architectures. However, we only mentioned a crucial part of the description: data; this is mostly because, while AADL provides some tools for data modelling (e. g., data components), it is not a fully fledged data description language. In a robotic system, since they are based on a *component-and-connector* paradigm, data play a key role to support the communication between components that happens, most of the time, through the exchange of messages or parametrised function calls. This role is so important, that most of the robotic frameworks and middleware develop their own language to describe data supporting communications, for example, ROS uses a simple message definition language to put the focus on the creation of custom messages when needed, the same happens for SmartSoft where the communication object are defined using a fully fledged Xtext-based DSL. There approaches exist so the designer of the middleware or framework can have full control on the shape and style of the message definition to correctly implement the low-level communication between components, however, this fragmentation makes the definition of a general purpose data modelling language quite challenging. Since AADL supports data components but it lacks tools for data description, the simplest solution would be to use data component as type identifier and adopt the target message description language as type descriptor. Considering that data components are

already software specific, this solution could work for everything related to communication, but it has two important limitations. First, while it works well for already existing messages that can be simply imported in the model, it removes any possibility of generality in the newly defined messages; an ideal solution would use a general description for custom messages that can be associated with different data type depending on the target platform. Second, communication-related data type are not the only type of data present in a system, a general data definition languages could be used to model all data in the architectures.

Beside of communications, there is another situation where data definitions play a key role: parametrisation of the component. When implementing a component there is a series of values that are calibrated for a specific configuration of the robot (e. g., maximum acceleration, camera resolution, wheel diameter, etc.), additionally, some of these parameters may need to change at runtime, both for on-line calibration (e. g., measurement weight during sensor fusion, obstacle avoidance threshold, etc.) and for dynamic functionality change (e. g., indoor vs outdoor operation, high vs low performance, etc.). In practice, different technological solution are used to implement parametrisation. ROS relies on a centralised component where all node parameters are stored and categorised using node-specific namespaces, at any time during execution a node can query the parameter server and retrieve a copy of the value. There is no strict differentiation between initialisation and runtime parameters, although ROS provides a separate system[3] to perform dynamic reconfiguration. Values in the parameters server are set or via command line before running the node or by loading a YAML file. SmartSoft uses a more complex structure for parameters, there are configuration parameters that are set at deployment time and cannot be changed after the component initialisation, and there are runtime parameters that can be used to configure the component at runtime. Definition of parameters is divided in two categories, too, they can be internal, therefore defined together with and specifically for the component, and external, thus defined as separate parameter sets

3 http://wiki.ros.org/dynamic_reconfigure

that can be reused in multiple component. The description itself of the structure of the data is done using a DLS similar to the message definition DLS. In this case there is no simple cohesive approach that can be used to easily import the existing definitions, because parameters, as a concept, are less standardised than communication protocols.

There is one last use of data in a component that is interesting to analyse under the lens of a data modelling approach: internal variables. Components are, in the end, computer programs, thus they may have tens of variables storing any type of value necessary for a successful execution. Of course it is pointless to try to model beforehand all the variables involved in the execution of a component, however, as we described in Section 4.1, there is a specific component behaviour, the *filter* with memory, where the type of information stored in the shared memory is defined ad design time, because it is part of the description of the functionalities of the component. For example, let us take a component in charge of obstacle avoidance, the designer knows in advance that the component will store the map of the environment, or a multiplexer component needs to store locally the inputs before relaying them. This design approach is similar to the object-oriented programming paradigm where before starting the implementation the developer design the class with all the necessary attributes and methods. By describing ports and components behaviours, we already defined the component-equivalent of methods, thus it is but a short step to fully embrace this consolidated design approach by defining the component-equivalent of attributes, too. Currently, no middleware or framework, not even those leaning more towards a model-driven development approach (i. e., SmartSoft and Orocos/RoCK) support the definition, at design time, of internal variables of the component. The reason of this lack of support with respect of parameters lies in the difference of complexity between the two; usually parameters are basic types (e. g., string, integer, boolean, etc.) or basic data structures (e. g., record, array, list, etc.), while internal variables can be any kind of object or data structure coming from internal or external libraries, moreover, often they require special initialisations. Unfortunately, given this complexity, we found no reasonable way to preserve

Listing 4.6: ROS message, service and action definition using ASN.1

```
CustomPackage DEFINITIONS AUTOMATIC TAGS ::= BEGIN
    IMPORTS Pose FROM Geometry_msgs;
    CustomMessage ::= SEQUENCE {
        x INTEGER(1 .. 10),
        y REAL,
        pose Pose
    }
    CustomService ::= SEQUENCE {
        request SEQUENCE {
            a INTEGER(-10 .. 200),
            b SEQUENCE (SIZE (1..10)) OF INTEGER
        },
        response SEQUENCE { sum INTEGER }
    }
    CustomAction ::= SEQUENCE {
        goal SEQUENCE { pose Pose },
        result SEQUENCE { success BOOLEAN},
        feedback SEQUENCE { pose Pose }
    }
END
```

the semantic of the internal variables in the same way as we could do for parameters. Nevertheless, we designed a compromise that maintains the semantic for simpler variables (i. e., basic types), but, at the same time, it allows for more complex types to be defined at design time.

These are all the situations in which data modelling is integral part of the design of a robotic application: communication objects, parameters, and internal variables. In the reminder of this section we will cover the two options we explored to be able to model all these features, their specific advantages and their limitations.

OPTION 1: ASN.1

Abstract Syntax Notation One (ASN.1) is an interface description language, it is a broadly used standard in telecommunication and computer networking. The language put a lot of focus on encoding and decoding and it is designed to be completely independent

from any computer or programming language. For these reasons, we considered it as our first option when trying to model the data exchanged in a robotic architecture.

In ASN.1 a communication protocol is defined in a module, inside, each element of the protocol (e. g., requests, responses, errors, etc.) is defined using a type. Normally, types are instantiated by a protocol data unit (PDU), however, when describing communication patterns, we use only defines modules (i. e., protocol descriptions) since the actual low-level communication is then left to the communication layer of the target middleware or framework. Nevertheless, this general description based on ASN.1, could be used to create a corresponding PDU and, eventually, an inter-protocol communication between different platforms. Listing 4.6 shows an example on how to model a ROS package containing all the possible user-defined communication objects, in particular a message, a service and an action. In this description, an ASN.1 module correspond to a ROS packages, this binding exists for multiple reasons: first, when importing an existing definition (Pose in Listing 4.6), the finest granularity available is the type defined in a specific module, this mirror the behaviour of messages defined in packages, second, modules are the only aggregator of ASN.1, therefore they have the same conceptual functionality of packages, lastly, types are the definition referenced by PDU, in the same way as message instances reference to a message definition in a package.

Messages are the most straightforward to define, since they do not have any internal structure other than the message itself. The are defined directly using a ANS.1 type and the fields uses the same basic types expected in a normal ROS messages, value ranges (e. g., `INTEGER(1 .. 10)` and `INTEGER(-10 .. 200)`) can be used to automatically identify the correct size of the target type (e. g., `uint8` and `int16`). Arrays can be defined using the keyword `SEQUENCE OF`, eventually specifying a minimum and maximum size. One of the characteristics of ROS messages is their hierarchical definition, they can always include other messages defined in the workspace, one typical example is the header used to timestamp messages. This is possible in ANS.1 using the `IMPORTS` keyword at the beginning of the module definition, and then declare a field

of a type to be of the imported definition. Of course this raises the problem of creating all the ANS.1 definitions for ROS messages, however, given the simplicity of the message definition language used in ROS, it is a task that can be easily automated to generate all the necessary files for the entire workspace. Internally, service definition is identical to message definition, the same rules are used to define basic types, arrays an to include existing types. The difference is in the structure of the ANS.1 type, for service it is necessary to differentiate between *request* and *response*, in the former, the developer describes the content of the request message sent by the client to the server, the latter contains the expected answer. The description of actions is similar, in this case the type is divided in three subsections: *goal*, *result*, and *feedback*. Each subsection can have an arbitrary number of field defined as basic types or existing types from other packages. For both services and actions, it is possible, in ROS, to send empty messages, for example a service used to trigger a functionality has an empty request, but the result of the functionality as a response. Empty sequences are supported by ASN.1, therefore, the correct way to model a service or an action with empty interactions is to define the subsection but leave it empty, as seen in Listing 4.6 for the feedback of the action.

To model the internal state of the node we have to follow a different approach. In this case, it is necessary to define both the module (i. e., the structure of the internal state) and the PDU (i. e., the specific configuration of the internal state), however these two specifications will be used at different times in the design, development and deployment cycle of the architecture. As for the communication objects, everything resides in the same ASN.1 module, but in this case, each module represent a specific node, although, if two nodes share the same parameters and variables configuration, they can share the same module (e. g., different implementation of the same functionality). Listing 4.7 shows how an hypothetical internal state of a node could be modelled using ASN.1, while Listing 4.8 represent a possible instance. The list of parameters and variables are defined each in their own type. For parameters, the description is quite simple, since they only use basic types and, potentially, nested records. Basic types are

Listing 4.7: Internal state of a node modelled using ASN.1

```
InternalState DEFINITIONS AUTOMATIC TAGS ::= BEGIN
    Complex ::= SEQUENCE {
        type UTF8String,
        include UTF8String
    }
    Parameters ::= SEQUENCE {
        size INTEGER DEFAULT 1,
        dimensions SEQUENCE {
            height REAL DEFAULT 1.0,
            width  REAL DEFAULT 2.0
        }
    }
    Variables ::= SEQUENCE {
        counter INTEGER,
        map Complex
    }
END
```

Listing 4.8: TODO

```
params Parameters ::= {
    size 2,
    dimensions { height 2.0, width 3.0 }
}

value Variables ::= {
    counter 0,
    map {type "costmap_2d::Costmap2D", include "costmap_2d.h"}
}
```

matched directly with their ASN.1 counterparts, again the range can be used to automatically specialise type during implementation or to define boundaries for the parameters. Often, parameters related to the same functionality (e. g., configuration values of a planner) are grouped together in a record data structure (e. g., C struct), to achieve the same result it is possible to declare the parameters as nested `SEQUENCE`. For most of variable definitions the process is the same as parameters, it is possible to use basic types and nested data structure to define and organise basic variables. The difference between the two lays in what we call "complex" variables, here the developer is not forced to use only basic types, but can use data structure of unpredictable complexity (e. g., ROS messages for storage, maps, point clouds, behavioural trees, etc.). It is unreasonable to try to capture this complexity in an high level design specification, therefore we defined an extra ASN.1 type called `Complex`. This type has two fields: `type`, the actual type of the variable in the target programming language (e. g., `costmap_2d::Costmap2D` for a ROS costmap in C++), and `include`, the resource to be included to use the specific data type (e. g., `costmap_2d.h`). The example and the structure presented here is specifically targeted for a C++ implementation of a ROS node, but the concept of the complex variable, can be easily translated to different programming languages. From a design point of view, the variable act as a conceptual placeholder for a more complex implementation, while during code generation the complex description can be used to automatically create the internal state of the component. For the description of the internal state, we can combine the module with the PDU to create a complete definition of the node. While in the module we can define default values for both parameters and variables, their actual values are going to change significantly depending on the particular deployment. To capture this, we can use the PDU to specify the current instance of the internal state, and automatically generate a specific initialisation for parameters and variables; complex variables, of course, are the exception, since the information contained in the PDU are used to define them during code generation.

Listing 4.9: ROS message definition using JSON schema

```
{
  "$id": "custom_package/CustomMessage.schema.json",
  "type": "object",
  "properties": {
    "x": { "type": "integer",
      "minimum": 1, "maximum": 10 },
    "y": { "type": "number" },
    "pose": { "$ref": "geometry_msgs/Pose.schema.json" }
  }
}
```

Listing 4.10: ROS service definition using JSON schema

```
{
  "$id": "custom_package/CustomService.schema.json",
  "type": "object",
  "properties": {
    "request": {
      "type": "object",
      "properties": {
        "a": { "type": "integer",
          "minimum": 1, "maximum": 10 },
        "b": { "type": "array",
          "minItems": 1, "maxItems": 10,
          "items": { "type": "number" } }
      }
    },
    "response": {
      "type": "object",
      "properties": { "sum": { "type": "integer" } }
    }
  }
}
```

Listing 4.11: ROS action definition using JSON schema

```
{
  "$id": "custom_package/CustomAction.schema.json",
  "type": "object",
  "properties": {
    "goal": { },
    "result": { },
    "feedback": { }
  }
}
```

OPTION 2: JSON WITH SCHEMA

JavaScript Object Notation (JSON) is an open-standard file format, it is human readable text where objects are codified in attribute-value pairs and array data types. While it was originally derived for JavaScript, hence the name, it is a language-independent data format, it is very common and massively used for asynchronous browser-server communication. While JSON is excellent to describe object instances (i. e., the actual data content), it is normally used with the assumption that the data structure is codified somewhere else (e. g., in the source code defining the object). However, there are various initiatives to define the structure and the content of the data using JSON, the one we are using in our approach is called JSON Schema; it is a vocabulary to annotate and validate JSON documents. The capability offered by the schema definition, combined with the extreme popularity of JSON made it our second option for data modelling.

A complete description based on JSON schema is composed by the schema, created following the vocabulary, and an instance of the data. Following a similar approach to ASN.1, for the description of messages, services and actions, we are going to define only the schema, and leave the instance to the underlying communication protocol. As before, the general description defined using JSON schema, can be used to create JSON documents with the same semantic value of the messages exchanges by the underlying middleware or framework, this is extremely useful since

JSON is one of the most common standard for the web, therefore, through this description, it is possible to create a web-compatible interface for the robot. Differently from ASN.1, each schema is not a complete package, but a single message, service or action definition, the same as it happens in ROS. To model the concept of packages it is possible to follow various routes, for example, locate the schema in the same folder, in the same way as ROS does, or exploit the unique id (`$id`) to categorise the schema, in our approach we use both. Listing 4.9, 4.10 and 4.11 show three different schemas representing a message, a service and an action, all of them defined in the same ROS package; physically, these three description are defined in three separate files and are stored in the same folder, exactly as it would happen for ROS definitions in the same package, moreover the `$id` is structured to specify both the package and the name of the definition.

As before, messages are the simplest to model, since they do not have any nested structure, and if they do, it is based on a different type. With JSON schema it is possible to directly define the list of field of the message and specify their type. The available basic types are: `string`, for a string of character of a specified length eventually matching a pattern, `number`, any type of real number where it is possible to specify the boundaries, `integer`, a subset of integer numbers, `boolean`, for a filed that can either be true or false, and `null`, for empty fields. Additionally a field can be of type `array` when it represent an array, in JSON there is no restriction on the type of the elements of an array, therefore JSON schema supports multiple options ranging from all the elements having the same type to each element has a different type. In ROS, only same-type array are admitted, in Listing 4.10 we modelled an example of a 10-elements integer array. JSON schema supports referencing external schemas, this can be used to model how in ROS message field can be of the type of a previously defined message. Additionally, most JSON schema validators support the use of URL as `$id` and `$ref`, this means it is possible to use the online location of a schema definition as the id and then use the same location during validation. In summary, only the custom message schema has to be parsed directly, everything else can be remotely checked only when necessary. As always, this raise the problem of generating

Listing 4.12: Base schema of the internal state defined using JSON schema

```
{
  "$ref": "#/InternalStateBase",
  "InternalState": {
    "additionalProperties": false,
    "type": "object",
    "properties": {
      "Parameters": {
        "type": "object",
        "additionalProperties": true
      },
      "Variables": {
        "type": "object",
        "additionalProperties": true
      }
    }
  }
}
```

the schema of all the existing messages, but not only this can be done automatically, they can be collected in a single online library (e. g., ROSWiki) or hosted in the same repository together with the ROS source code. Regarding the description of types, services are defined in the same way as messages. They can use basic types, arrays or reference existing messages. However, their schema is divided in two section defined as two different objects: *request* and *response*. As before, the former describe the message sent to the server to trigger the service and the latter models the answer received by the client. The same structure based on sub-objects is followed by the action description, this time divided in three different parts: *goal*, *result* and *feedback*. Each subsection follows the same rules of the message, can have multiple field as basic types, arrays or reference existing definitions. One significant difference between JSON schema and ASN.1 is how they describe empty messages. In JSON schema empty brackets (i. e., *{ }*) represent the wildcard (i. e., it matches every sequence), the correct way to specify an existing, yet empty, field is to declare it as type `null`, as seen in Listing 4.11 for the feedback of the action.

Listing 4.13: Internal state defined using JSON schema

```
{
  "$ref": "#/CustomInternalState",
  "InternalState": { "type": "object",
    "properties": {
      "Parameters": { "type": "object",
        "properties": {
          "dimensions": { "type": "object",
            "properties": {
              "height": { "type": "number", "default": 1.0 },
              "width": { "type": "number", "default": 2.0 }
            }
          },
          "size": { "type": "integer", "default": 1 }
        }
      },
      "Variables": { "type": "object",
        "properties": {
          "counter": { "type": "integer" },
          "map": { "$ref": "#/complex" }
        }
      }
    }
  },
  "complex": { "type": "object",
    "additionalProperties": false,
    "properties": {
      "type": { "type": "string" },
      "include": { "type": "string" }
    }
  }
}
```

Listing 4.14: Internal state instance defined in JSON

```
{
  "Parameters": {
    "dimensions": { "height": 2.0, "width": 3.0 },
    "size": 2
  },
  "Variables": {
    "counter": 0,
    "map": { type "costmap_2d::Costmap2D", include "costmap_2d.h" }
  }
}
```

As mentioned before, when describing the internal state we have to use both the instance, representing the actual values of the parameters and variables, and the schema, the model that validates the structure of the instance to ensure its compatibility with the component. When using JSON schema we opted for a hierarchical approach based on a double validation. First, we validate the instance to verify the general structure, and then we use the component-specific schema to validate the content. Listing 4.12 shows the schema used for the first step of the validation. The designer has to follow this model, but he does not have to implement or provide it, since it is embedded in the validation and code generation process. The base schema is used to verify that the instance follows the structure where the internal state is divided in two, and no more (see `additionalProperties` set to false), subsections: parameters, values that are initialised at the beginning of the life cycle of the node and are not subject to changes, and variables, values used to capture the evolution of the internal state of the component. Listing 4.13 represent the schema of an hypothetical internal state of a component, and it is the definition used in the second step of the validation process. Following the structure enforced by the base schema, parameters and variables are defined each in their own object. Since parameters are defined using basic types and nested records, their description is quite straightforward. JSON schema types are used to define their implementation-specific counterparts, while the `object` type

can be used to define nested records. For variables definition, the process is mostly the same, basic types variables are defined directly using JSON schema types, and it is possible to create nested structures to organise them. Similarly to ASN.1, we defined an new object to capture "complex" variables. This object has exactly (enforced by setting `additionalProperties` to false) two values: type, defines the actual complex type of the variable, and include, in a C/C++ specific implementation it provides the resource to be included to correctly use the complex type. The *complex* object can be redefined to match the requirements of different programming languages, for example, Python does not require a type but may need to specify which resource to import. The schema presented in Listing 4.13 represents only half of the necessary description to completely model the internal state of a node; it is the static design and it is completed by the JSON instance. An example of an instance compatible with the two schemas presented is in Listing 4.14.

COMPARISON

Both ASN.1 and JSON schema are powerful enough to completely capture the description of communication objects and the internal state of the components, therefore they are almost interchangeable as data description language. Additionally, ASN.1 supports JSON encoding rules (JER), so, in theory, it would be possible to create a direct conversion between the two different representations. All considered, it may not be necessary to pick a specific option to model the data in the architecture, however, there are some reasons that make one approach more suitable than the other. ASN.1 is a widely used protocol description language that supports multiple encodings and can deal automatically with various low-level communication issues (e. g., endianness, payload size, etc.), this is extremely useful when modelling communication with physical devices, therefore it can be used when dealing with sensors and actuators. However, ANS.1 can be too low-level for most applications and while widely used it is not a widely known language. JSON, on the contrary, was created to encode JavaScript objects, thus it does not capture low-level details, but it is a de facto

standard for web communications and NoSQL databases. This means it can be used to create communication bridges between different technologies (e. g., ROS and SmartSoft), web APIs for robots or advanced logging systems.

In summary, both approaches are suitable for modelling all the data exchanged in robotic systems, but the specific characteristics of the languages make them more or less useful when facing specific problems.

5 AUTOMATIC PROGRAMMING

Writing machine code involved several tedious steps—breaking down a process into discrete instructions, assigning specific memory locations to all the commands, and managing the I/O buffers. [...] We needed to understand how we might reuse tested code and have the machine help in programming. [...] This led to the development of interpreters, assemblers, compilers, and generators—programs designed to operate on or produce other programs, that is, automatic programming.

— Mildred "Milly" Kross

Creating a computer program is a challenging and engaging experience, it requires expertise, brilliance and ingenuity. At the same time, writing code is a mundane and dull activity, it requires to complete repetitive tasks, to manage multiple small issues and to deal with problems unrelated with the main project.

Thankfully, today, we are not dealing with the same difficulties that Mildred Kross faced when working on the UNIVAC I. Multiple technological advancements and progresses in Computer Science gave us compilers, high-level programming languages, design paradigms, frameworks, middleware, integrated development environments, and more. All these tools exist to make computer programming more about designing and developing a program than writing code.

During Computer Science history, the concept of automatic programming changed to adapt to the expectation of the time. Originally, it was the automation of the process of punching paper tape, later it became the transformation of high-level programming languages (e. g., Fortran and ALGOL) to machine code, a task that, nowadays, is integral part of the build process. Today, automatic programming mostly refers to the automatic generation of executable code from representations that are not programming languages. These representation may have different level

of abstraction, from the closest to the target (e. g., domain specific languages, flowcharts, etc.) to the furthest (e. g., graphical representations, models, etc.).

Here we present our approach to automatically generate a ROS architecture from an AADL model combined with a data description model. The chapter starts with a description of our target, specifically, an engineered version of a ROS nodes including advanced functionalities. Then the automatic programming process is presented with its two-steps approach. The chapter closes with an example going from model to code.

Contents

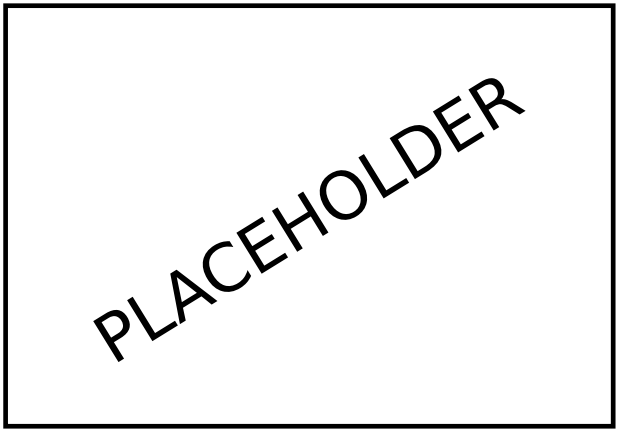


Figure 5.1: TODO

5.1 GENERATING ROS ARTEFACTS

An automatic programming approach is a collection of rules and methods to transform an initial description to a different, more complex output. Therefore, even before the definition of the transformation itself it is necessary to define the input and the output of the process. In Chapter 4 we described in details a collection of meta-models that can be used to define ROS architectures using AADL in combination with a data modelling language (i. e., ANS.1 or JSON schema). A model compatible with this meta-models is the starting point of our automatic code generation process. Since the expected output of the entire process is a complete working architecture, the model alone is not enough as an input. To have a functioning architecture, the designer needs to pair the model definition with the implementation of the node inner functionalities; this can be done by using AADL properties. A ROS complete architecture is composed by multiple elements: existing nodes run as external resources, new nodes and messages created by the developers, launch files to organize the architecture, parameter profiles to configure the system. All these elements are captured by the model, and can be automatically generated by our process. First, the automatic programming system creates

the source code for the new custom nodes, the target language is C++. ROS supports multiple languages, mainly C++ and Python, Lisp is officially in the list but rarely used, and there are experimental libraries for Java and Lua. From the ROSwiki, *rospy* (i. e., the Python implementation of ROS) is suggested as the approach that promotes implementation speed (i. e., reduced development time) over runtime performance, and it is designed specifically for fast prototyping, testing and lightweight implementations (e. g., configurations and initialisations), while *roscpp* (i. e., the C++ implementation of ROS) is considered the main library, and it is designed with a focus on high performances and runtime speed. Since the output of the automatic programming is at the end of a long process involving a model-based design and carefully development components, we decided to use *roscpp*, and therefore C++, as the target library to achieve the most efficient and robust implementation. Since C++ is a compiled language, the system will automatically generate all the necessary files to build the node executables; if the designer specify all the necessary information in the model (i. e., source code of the functionalities), the final output of the automatic programming process will be ready to compile with no intervention required. The automatically generated code will be placed in the correct package structure expected by ROS, together with any custom message, service or action file. This covers everything necessary for the execution of single nodes, to put them together in an architecture it is necessary to create launch files. In launch files, the designer specifies the instances of the nodes and how they are connected, by renaming all the necessary topics, and configured, by including configuration files. The topology of the architecture can be automatically extracted from the model and converted in a launch files, and the parametrisation defined using a data modelling language (i. e., ANS.1 or JSON schema) can be converted in the YAML description used by ROS. Moreover, in launch files existing nodes are included in the architecture and connected to the rest of the topology.

Figure 5.1 summarises the complete process. Our automatic programming approach requires as input a model defined in AADL, completed by a data description using ASN.1 or JSON schema, and specialised via properties to include functionality-

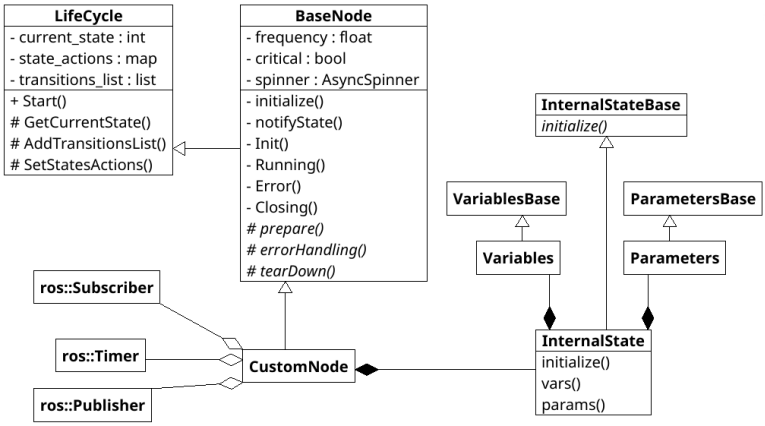


Figure 5.2: UML diagram of a custom ROS node developed extending the engineered ROS node.

specific source code. When all these conditions are met, the process provides as an output a collection of automatically generated and compilation-ready ROS nodes, their associated communication files (i. e., messages, service and action files) and the necessary launch files to run the architecture. A fully complete model creates an architecture that only needs to be compiled and run.

5.2 ENGINEERED ROS NODE

Differently from other middleware or frameworks, ROS does not constrain the developer on the structure of the components; it was designed to be maximally flexible following the mantra: "*we don't wrap your main()*". While this approach certainly contributed to the popularity and growth of ROS as a middleware and de facto standard for robotics, at the same time it created a very heterogeneous landscape for ROS nodes. Some of them are well designed, rich in functionalities, robust and configurable, others are cobbled together for a prototype and then used as legacy code for one single core functionality. Often this second category is created by

experts in a specific field (e. g., vision, control, manipulation, etc.) that lack the necessary programming and software engineering skills and knowledge to develop a well-designed and robust node. With our approach we are targeting specifically this category of developers, that posses the expertise to contribute to the robotic community, but are discouraged by the programming required.

Since ROS does not impose any structure for the node, the simplest approach for automatic code generation would be to target an essential node, covering the minimum functionalities required to run it. There are few advantages in this approach: easier to implement code generator, simpler and more readable output, an implementation closer to what the developer knows. However, such a direct approach would have significant downsides: a lax relationship between the node and its model, lack of advanced functionalities that can be hidden in the automatic programming approach, less flexibility of the implemented node, more work left to the developer (e. g., testing, debugging, performance evaluation, etc.), more tampering by the developer with the basic structure of the node, no real benefit between a handcrafted and an automatically generated node. For all these reasons we decided to created an engineered base node that can be used as a reference and starting point for automatic code generation.

Figure 5.2 shows a UML diagram of a custom node based on the engineered ROS node. Immediately, it is possible to recognise three main components: *LifeCycle*, *ROSNode*, and *InternalState*. Each of them represent one of the main characteristics captured by the engineered node. The *LifeCycle* implements an internal state machine that controls the evolution of the node. The *ROSNode* is the core implementation of the node, capture all the ROS-related functionalities and management procedures. The *InternalState* capture all the developer-defined parameters and variables necessary for the correct execution of the node.

LIFE CYCLE

When working with component-based approaches, it is important to define a recurring and consistent behaviour of the component, in the case of robotic components it is even more important, since

they often operate with strict timing constraints and implement critical functionalities. In Section 4.3.1, we presented how a life cycle of a node can be modelled in AADL, and how it can be used to guide the initialisation, configuration and execution of a component. To capture the same behaviour in the engineered node, we developed the *LifeCycle* class to define the evolution of the node. While various implementations of state machines in C++ already exists, a very popular one has been around for almost 20 years[1], we opted to create a stripped down version that trades some functionalities for simplicity, understandability, and modern development approaches. The result is a very lightweight state machine, that supports dynamically defined states and transitions and it is completely ROS-independent.

To maintain the generality of the implementation, the class itself does not specify any state or transition. It only defines an empty enumeration that the subsequent classes can extend to define their own states. The valid transitions are defined as a list of pairs, going from one state to another, as for the states, the list is created by classes extending or using the state machine. The last initialisation step before running the state machine is to bind each state to a function. In practice, the binding is done by creating a map with the state as the key and a `std::fuction` as the value. Class template `std::function` is a general-purpose polymorphic function wrapper, it can be assigned to any callable target (e. g., functions, lambda expressions, pointers to member functions, etc.), this makes this approach particularly flexible and not bound to any specific implementation. When the initialisation is complete, the state machine can be started, the initial state is the one defined in the constructor, but there is no specific definition for a termination state. At each execution loop, first, the state machine execute the function bound to the specific state, then, it checks if there is a valid transition waiting to be performed, if there is one, the loop repeats and a new state-bound function is executed, otherwise the state machine has reached a final state and the execution terminates. While the list of all possible transitions is defined during the initialisation phase, each specific change

1 https://www.codeproject.com/Articles/1087619/State-Machine-Design-in-Cplusplus-2

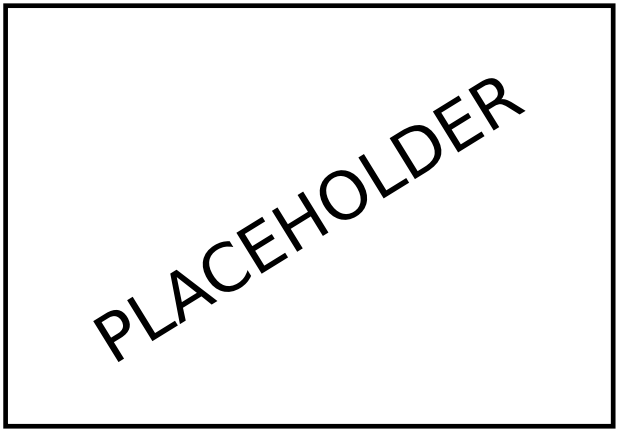


Figure 5.3: TODO

of state is defined at runtime in all the state-bound functions. This is necessary, for example, to distinguish between a successful component initialisation that goes from an initial to a running state, to an unsuccessful one that would take the component to an error state. In practise, at the end of each state-bound function, the developer needs to specify the next state depending on the current outcome of the execution, then the state machine will check if the transition is valid (i. e., it is in the transitions list) and execute it. Mirroring the model presented in Section 4.3.1, the engineered node supports five different states: initialisation, running, error report and closing. How they are implemented, what is their role and which ROS functionalities they evoke will be detailed in the next section.

ROS NODE

The core implementation of the engineered ROS node is in the *ROSNode* class, here the life cycle is defined and materialised, and all the basic ROS-related functionalities are implemented. By defining this class, we can streamline the development process of a component by hiding the base initialisation procedure of a ROS node, create a well defined structure the developer can follow, and

enhance the base implementation by adding additional functionalities (e. g., error detection and state report). The *ROSNode* class extends directly the *LifeCycle* class, therefore, the first implementation step is to define the states, the valid transitions and method bound to each state. Figure 5.3 presents the complete definition of the internal state machine, this mirrors the description provided in Section 4.3.1. Each state is bound to a specific method of the class, and it implements a core functionality of the node.

INIT – This method is bound to the initial state of the node (i. e., `ST_INIT`). It is defined as a two-steps process and all the initialisation procedures of the component are implemented here.

The first step is a common initialisation that applies to every node, it sets up the ROS environment and defines the asynchronous spinner[2]. In Section 4.3.1, we modelled the ROS node with a external port to communicate the current state of the node after every transition, in this phase of the initialisation, the base node create the connection with the ROS service in charge of receiving this notifications. In ROS, a client can register to a service even if the server is not active, all the communications are lost until the service is finally started. This behaviour is not a problem for a status notification system, since it is meant to exist only to supervise the general evolution of the internal life cycle of the node. Nevertheless, our aim is to create a flexible base node that can adapt to different situations, thus, we defined an extra configuration property for the node, the developer can initialise the node as *critical*. When a node is critical, instead of just starting the status notification service, the initialisation procedure will wait until the service is up, and then registers to it. In this way, an external supervisor node in charge of monitoring all the critical nodes can trace the entire evolution of their life cycles and act accordingly if something does not behave as expected. After the general initialisation procedure, the first action is to externally notify the state of the node, again the notification method itself behave in a different way for critical and non-critical nodes. To save bandwidth and reduce the number of requests, non-critical nodes notify their new state after a transition only if it is differ-

2 http://wiki.ros.org/roscpp/Overview/Callbacks and Spinning

ent from the previous one, in other words, they do not notify transitions on self-loops. Critical nodes are meant to be constantly monitored by the supervisor, therefore they notify their state after every transition, basically, in this case, self-loops are used as a way to measure the liveness of the node.

If the basic initialisation is completed successfully, the second step of the set up of the node is activated. An unsuccessful outcome is considered a critical failure and the node is instantly shut down. The second part of the initialisation is an abstract method not implemented in the *ROSNode* class (i. e., the *prepare* method) and it is meant to be used by the child class to define any node-specific initialization procedures. The developer can use this method to fill parameters and variables with their initial values, set up publishers and subscribers, or perform any other special initialisation (e. g., activate hardware connections, pre-fill data structures, etc.). If this preparation phase is completed successfully the method will trigger the transition to the main execution state. Differently from the base initialisation, an unsuccessful preparation phase will trigger a transition in error state. This is because we cannot anticipate what kind of procedure the developer will implement in this second part of the initialisation, therefore issues in this phase may be resolved in a specific error management procedure and lead to a successful initialisation.

RUNNING – A complete and successful initialisation will transition the state machine in the `ST_RUNNING` state, and this is the method bound to it. Since the engineered ROS node is built around an asynchronous spinner, most of the ROS-related functionalities (i. e., checking subscribers, services and timers) are executed in a separate thread, hence this method only needs to check for errors or node termination. When the node is working with no issues, the *Running* method is just a low-frequency (i. e., 1 Hz) repeating self loop, two things can change this condition: first, one of the asynchronous activities (e. g., a subscriber callback) sets an error flag, or second, an external signal triggers the shutdown procedure. In the former case, this method interrupts its self loop and trigger a transition to the error state, in the latter, it receives the signal and changes the current state of the life

cycle to shutdown. When the node is initialised as *critical*, the behaviour of the *Running* method does not change, however, the state notification happens at every self loop instead of only after the first transition. Given the fundamental functionalities implemented in this method, the developer cannot directly modify it, however it is possible to change the frequency of the self loop. Given the structure based on the asynchronous spinner, changing the frequency will not influence the behaviour of any ROS-related functionality, but it can change how fast the node reacts to errors or terminations; specifically, the higher the frequency the shortest will be the time between the generation of errors and interrupts and their detection. Changing the frequency of the self loop it is also useful to monitor the liveness of critical nodes.

CLOSING – As any other process, ROS nodes can be closed by sending a termination signal (i. e., SIGINT). Normally, they already implement a handle to capture the signal and force a shutdown of the node, in the engineered node we replace it with our own version. Our handle does not perform any shutdown procedure, it just capture the signal and sets a flag; this flag will cause the *Running* method to trigger a transition to the `ST_CLOSING` state. In this method bound to the state is where the actual shutdown procedure happens: first, it triggers a final state notification, so a potential supervisor knows that the node is shutting down, then it execute a custom *tearDown* method, and finally it shuts down any ROS-related functionality. The *tearDown* method is the shutdown counterpart of the *prepare* method of the initialisation. It is abstract and not implemented in the base class, any child class extending *ROSNode* will implement it with their own specific shutdown procedure. This is done to let the developers gracefully close existing connections (e. g., device drivers), propagate the shutdown to other nodes (e. g., critically dependant components), or provide additional shutdown notifications (e. g., specific logging systems). Since this is the terminal state of the life cycle, this method does not implement any transition.

ERROR – Natively, ROS does not provides any system to manage errors during the execution of the node. In our engineered node, we created an extensible structure to identify, detect and react

to errors. The *ROSNode* class defines an enumeration with few predefined error codes, they cover some common issue that may happen during the normal functioning of a node. They are:

- `PARAM_ERROR`: this is a typical initialisation error, when a necessary parameter is not found in the parameter server. The node cannot run correctly when partially initialised, a way to handle this error is to wait until the parameter is available and then restart the initialisation procedure.
- `SUB_FAILED`: one of the subscribers of the node fails. This may compromise the entire node (e. g., set-point subscriber for a control node) or just disable one of its functionalities (e. g., a multiplexer missing one of the inputs). Depending on the situation this may or may not require the shutdown of the node.
- `PUB_FAILED`: one of the publishers of the node fails. As for the subscribers, this may be very impactful (e. g., a planner that cannot publish the result) or a minor inconvenience (e. g., a visualisation topic not initialised), and therefore could prompt a shutdown of the node.
- `INVALID_MESSAGE`: This is an error that can be triggered during a subscriber callback or before publishing a message. A message has an unexpected value (e. g., negative distance, out of bound acceleration, empty path). Sometimes malformed messages are inconsequential, and the error handler will just record them, in other cases they can be hazardous and require to halt the node execution.

These error codes are mostly related to the correct execution of ROS functionalities, on top of these, the developer can extend the enumeration and define his own error codes, to cover problem-related corner cases.

At any moment during the normal execution of the node, the developer can use the *faultDetected* method to notify one of the possible error codes. After the initialisation process or during the *Running* self loop, if any error code is set, the system will transition to the `ST_ERROR` state. The *Error* method is bound to this state,

during its execution, first it notifies the state transition, to let any potential supervisor know that the node is in an error state, then it calls a method to handle the error, finally, after *errorHandling* returns, if the error code is still set, the error is unsolvable and the node transition to the `ST_CLOSING` state, otherwise, the node can go back to normal execution. The *errorHandling* method is similar to the abstract method used during the initialisation and closing phases, as before, it is an abstract method that the developer can extend in the child class to manage any specifically defined error codes. Differently from the previous two, it is not a pure abstract method, since we provide a basic implementation in the *ROSNode* class to manage the four already defined error codes. The developer can decide to reimplement completely the method (e. g., complex nodes with articulated initialisation procedures), run it alongside the existing one (e. g., few problem-related corner cases), or skip error management and leave only the already defined implementation (e. g., simple node requiring minimal definitions).

INTERNAL STATE

Components are meant to be reusable through composition and parametrisation. The computation graph of ROS combined with our model-based approach ensure the composability, by providing deployment-time rewiring and decomposing components functionalities. On the contrary, parametrisation is not embedded in the design of the nodes. ROS provides various system-level tools to manage parameters: a centralised system to store and collect parameters (i. e., the parameter server), the corresponding APIs to fetch and set them, and a system to dynamically change parameters in real time. However, how to integrate them in the component is left to the developer. The result is parametrisation is often ignored or misused. Parameters are uncategorised and mixed between functionalities, modified during execution creating inconsistencies, defined conceptually but then hard-coded as constants in the implementation. This issues prompted us, originally, to exploit data modelling languages to capture the parametrisa-

tion of the component, and, in the engineered node, to design a separate class to encapsulate the parameters of the node.

Parameters are the external-facing part of the internal state of the component, they codify a specific configuration defined before execution; variables, on the opposite, are internal-facing, they evolve with the node and exist only in the time frame of its execution. As mentioned at the beginning of this section, ROS does not provide any structure to support the internal state of the node, in the case of the variables, it means there is no predefined way to safely store and share between callbacks the information extracted and derived from messages. Normally, there are two approaches used when developing ROS nodes, one is for traditional imperative implementations, where variables are declared as global and shared between callbacks, the other wrap the entire node in a class, it uses methods as callbacks and attributes for variables. While this approaches are not inherently wrong, they have significant downsides. They reduce the portability of the node design by removing the distinction between ROS-specific (e. g., declaration of publishers and subscribers) and problem-specific (i. e., the necessary logic to implement the functionalities of the node) implementations, they are harder to debug since they are not encapsulated and may have unpredictable side effects, they are more difficult to extend and modify since they do not present a common interface. Our solution is to create an overarching class encapsulating both parameters and variables, and present a single interface that the developer can use to access, modify, share and store the internal state of the component.

As visible from Figure 5.2, the internal state of the node is built using a hierarchical approach. The superclass is called *InternalStateBase*, it has two members: a structure for variables of type *VariablesBase* and a constant structure for parameters of type *ParametersBase*. Parameters are defined as constant, this means that after setting their values during the initialisation phase, it is guaranteed they will not change. An exception to this is when the node implements a dynamic reconfigure system, in this case there is a special callback in charge of managing the parameters and changing them according to an external control panel. In general, by defining the structure as constant it is possible to limit any

modification of the parameters to specific procedures, and avoid unintentional modifications using compile-time checks. Both the parameters and variables structures are defined as shared pointers, this pointer-based declaration gives us the ability to treat each structure as a single entity, while, at the same time, avoiding useless and memory-consuming copies when accessing them in external procedures. The superclass has only one method, a pure abstract initialisation method that the developer has to implement in the child class; this is designed to force the developer to initialise all the parameters in the same location and to set an initial value to all the variables. To increase the flexibility of the base node, the internal state is not declared directly in the *ROSNode* superclass, the developer can extend *InternalStateBase* to create his own internal state class and declare it in the custom node. To better exploit the structure provided by the superclass, the developer can extend the parameters and variables structures, to define all the problem-specific details.

5.3 CUSTOM ROS NODE

In Section 5.2, we presented the engineered node as the starting point for the automatic programming process, however, the classes defined are perfectly suitable to be used directly by a developer to create their own node. In this section, we will present how a custom node can be implemented using the engineered node as a starting point, using the same approaches and structures the automatic code generator would create.

Since it is an independent class used by the rest of the node, the first step should be the definition of the internal state, and in that, the developer has to start from the definition of his own variables and parameters structures. Although the two base structures do not provide any functionality, it is ideal to extend them and exploit a polymorphic approach to define the internal state. The automatic code generator extends them and adds constructors to initialise the instances to their default, for parameters, and initial, for variables, values. The *InitialStateBase* class exists to

provide an unified interface between the node and its internal state, therefore, following the concepts of information hiding used in object-oriented programming, it is necessary to implement accessors for both parameters and variables. Since we are working with pure data structures declared as shared pointers, the most elegant approach is to define accessors for the entire structures and let the developer use directly the fields. To complete the definition of the internal state, it is necessary to implement the abstract initialisation method. A developer can create any complex initialisation, but in our automatic programming approach, we delegate the parameters set up to the core node functionality and use a copy constructor, while for the variables we invoke the constructor with no parameters defined in the structure.

After completing the definition of the internal state, a developer can move to the implementation of the node itself. The engineered node wraps all the main ROS-related functionalities in a single class, to implement the custom node the developer has to extend the *ROSNode* base class. In the custom node class declaration three abstract methods are overridden from the superclass: *prepare*, *tearDown* and *errorHandling*. Additionally, all the callbacks related to subscribers, services, timers and actions are defined as class methods. Since the class is the container of all the ROS-related code, the timer, publisher, subscriber, client, server and action objects are all declared as class members. Lastly, to complete the class definition, it is necessary to declare the internal state instance. These are all the methods and attributes that are necessary to implement the child class as an extension of the *ROSNode* base class, and to cover all the fundamental ROS functionalities. When using the code generator only these methods and attributes will be automatically created, however, there is no restriction on the complexity of the class and a developer can declare all the additional helper methods and attributes necessary to ensure the proper functioning of the node.

As introduced in Section 5.2, the *prepare* method is meant to contain all the node-specific initialisation procedures. Here the code generator will create all the necessary calls to collect the parameters from the ROS parameter server and initialise the internal state of the node by setting the initial values of the variables.

Moreover, callbacks of subscribers are initialised, publishers and services are advertised, and timers are set. At the end of this method, the node is ready and fully functional. Since it is abstract, the developer needs to implement the *tearDown* method, here, all the necessary procedures to gracefully shutdown the node are executed. ROS manages all the connections transparently with respect to the end user, therefore, in a generic node there is no need for any particular shutdown procedure. If not specified otherwise in the model, the automatic code generator fills this method only with an output message to notify the system that the node is shutting down. The last method that is necessary to implement to complete the definition of the child node as an extension of the base class is *errorHandling*. All the structure to manage errors is not ROS-related and it is defined by us in the base class, since this is an extra functionality not necessary for the basic operation of the node, we already provide a simple handler that can be called directly and manages basic initialisation errors. However, if the developer wants to exploit the error management system, he can define his own error codes in the class by extending the enumeration defined in the *ROSNode* superclass, and implements his own version of the *errorHandling* method. Potentially, a developer could implements an extension of the internal life cycle of the node, where instead of a single error state there are multiple error states, each for a different category of problems. This can be done by extending state enumerator, binding new methods to the new states, and add all the necessary transition in the state machine. This is possible because by overriding the *errorHandling* method, not only the developer can develop his own error management procedures, but he can change how the internal state machine evolves from the `ST_ERROR` state. Without any specification in the model, the code generator will not override this method and use the implementation provided in the base class. While it is possible to specify in the model an implementation for the method itself, if the developer wants to change the life cycle of the node, he needs to do that by modify directly the source code, since the modifications are too radical and in depth to be managed by an automatic programming system.

The last step to complete the implementation of a custom node is to define the logic related to the subscribers, timers, or actions callbacks. As mentioned at the beginning of this section, all the callbacks are defined as methods of the child class; ROS already defines the signature for all the callbacks, hence the developer only needs to implement them. He can read parameters from the internal state and store the output of processing in the variables structure. Practically, there is no issue in implementing the logic directly in the callbacks, however, with our automatic programming approach we try to push as much as possible the separation between the problem-specific and the ROS-related implementation. The automatic code generator creates a function call for each callback using the structure defined in the model as a reference. The functions have access to the parameters and variables structures and to the message, but their entire implementation is completely independent from the node. This function can be used as a bridge between ROS and a domain-specific library and can be easily tested and debugged without the need of running the node. With the addition of the logic the implementation of the custom node is complete. Of course, since this is a C++ implementation, the developer needs to create all the necessary configuration files to build the executable. With the automatic programming approach, all the necessary file are generated, and the developer only needs to add additional sources that were not specified in the model.

5.4 TWO-STEPS CODE GENERATION

Now that we have completely defined the input, a complete model description defined in AADL following the meta-model presented in Chapter 4 combined with data modelled using ASN.1 or JSON schema, and the output, the engineered ROS component implementing advanced functionalities and a list of approaches to implement a custom node, we can describe all the necessary transformations to convert the collection of model to a working architecture. In our toolchain, we adopted a two-steps approach,

first a model-to-model transformation that converts the input AADL model to an intermediate XML-based representation, then a model-to-text transformation to automatically generate ROS-compatible C++ code.

While AADL is popular in some specific fields (e. g., space applications, automotive and embedded hardware), it is a niche language, therefore there are only few options in terms of tools and support. In particular, there is only one maintained and open-source AADL model processor that supports model checking, parsing and code generation: Ocarina. As presented in Section TODO, Ocarina is written in Ada and provides multiple functionalities (e. g., parsing, model analysis, schedulability analysis, etc.), but mainly it is a parser and code generator. With its frontend/backend structure it separates the parsing and syntax analysis of the AALD model (i. e., frontend) from the code generation of a specific target (i. e., backend). This means that a developer could implement its own backend to exploit the parsing capability of Ocarina to create his own code generator. In theory, we could have used this approach to create directly a code generator completely implemented as a backend, but we decided to use Ocarina only to create an intermediate representation.

There are multiple reasons behind this choice. First of all, not only Ocarina is implemented in Ada, but is follows the Ravenscar profile, this is necessary because some of the code generation targets are certified for safety-critical hard real-time applications, therefore the toolchain itself needs the same certification. The Ravenscar profile imposes some restrictions on the already challenging Ada language, making the development and maintenance of a new backend a difficult and time-consuming task. By using an intermediate representation we can implement the most challenging part of the code generation (i. e., ROS source code) with our preferred approach. Additionally, using an XML-based language, that we called AAXML, creates an intermediate artefact that can be used as a starting point for code generation, but also as the output of a different AADL parser. Ocarina is an independent project that we do not control directly, therefore we cannot base our entire toolchain on a technological solution that could disappear or change drastically. Lastly, a two-steps approach makes

the entire toolchain more flexible, for example to extend the code generator to support ROS2 or a different framework, or to include additional models (e. g., a specific language to describe the component behaviour). There is one final, more practical reason, to adopt a two-steps approach and instead of generating directly ROS/C++ code, Ocarina already implements a backend that uses XML as a target. Unfortunately, the existing XML backend is currently non functional and not maintained, it was developed as a low-priority approach in an effort to create a bridge between AADL and a possible XML-based representation. Nevertheless, we used it as a guideline to create our own backend from scratch.

FROM AADL TO AAXML

This is the first step of the code generation approach, it is a model-to-model transformation, since the original structure of the AADL model is preserved but converted in an XML-based representation called AAXML. The Ocarina frontend provides two output that can be used by the backend: an abstract semantic tree, this guarantee correctness of the syntax and semantic of the model, and an instance tree, this is the result of the instantiation process; both of them are used by the backend. To implement the *aaxml_ros* backend, we followed the structure already established by Ocarina, with few modification to make the toolchain more suitable to the needs of a robotic system. First the backend needs to be registered to the frontend, so it can be called by the toolchain, after this simple initialisation phase, the automatic code generation process can start; it is a recursive approach that analysing the root system of the model and goes through subcomponents, features, connections, and properties. In the default Ocarina implementation of a backend, only a single system can be parsed by the toolchain per call, but since in our model the hierarchy of AADL systems represents the deployment configuration of the architecture (i. e., launch files in ROS), we decided to modify the backend to parse all the root system component (i. e., system that are not subcomponents of other systems) to be able to generate all the necessary launch files at the same time.

COMPONENTS – As mentioned before, the code generation process is recursive, it always starts from a component. A component is mostly characterised by its subcomponents, ports and connection, however, few details can be extracted directly from it and converted to AAXML. The *name*, an unique identifier of the actual instance of the component, necessary when there are duplicates coexisting in the same architecture. The *type*, the reference to the specification of the component, depending on the level of specification it can be an interface or a complete definition. The *category*, this attribute specify the AADL type (e. g., process, thread, system, etc.) associated with the component, it is guarantee to be compatible with the container component by the syntactic and semantic analysis done by the frontend. The *namespace*, it specifies the package containing the component, while this seems a superfluous information, it is extremely important when referencing component not defined in the same package of the system. The subcomponents list, AADL has a hierarchical structure and Ocarina manages it using a recursive approach, this list is used to perform all the recursive call and complete the traversal of the tree.

Listing 5.1 shows a small example of a system containing a single process as subcomponent, this is the minimal AADL architecture that we can model and instantiate. In Listing 5.2 presents the AAXML counterpart of the model, where the properties described are encoded in an XML-based format. In this minimal structure, no information is lost when converting the model from AADL to AAXML.

Listing 5.1: TODO caption

```
system root_system
end root_system;

system implementation root_system.impl
    subcomponents
        main_process: process custom_process.impl;
end root_system.impl;
```

Listing 5.2: TODO caption

```
<system>
    <type> root_system.impl </type>
    <category> system </category>
    <namespace> aadl_xml </namespace>
    <subcomponents>
        <component>
            <name> main_process </name>
            <type> custom_process.impl </type>
            <category> process </category>
            <namespace> aadl_xml </namespace>
        </component>
    </subcomponents>
</system>
```

FEATURES – The subcomponents list is one of the defining characteristics of a component, the other is the set of features. They define the frontier between the component and the external environment, moreover, when working with interface-only definitions (e. g., existing ROS nodes), they are the only element characterising the component. For these reasons it is important to capture all the necessary information when converting the model from AADL to AAXML. Features have a list of characteristics that are mandatory and are necessary to specify them, and others that optional. First of all, it is necessary to identify the type of feature, AADL categorises them in two main groups: accesses (i. e., subprogram, data and bus) and ports (i. e., data, event and event data); this is captured by the *category* tag. By defining the *type* it is possible to specify the specific subcategory of a port (e. g., *event_data*), for accesses there is no need for specialisation since the component they are connected to determinates their specific subcategory. Another attribute that is unique to ports and not necessary for accesses is the *direction*, since ports can define a specific ingoing, outgoing or bidirectional communication. While in the model it is necessary to specify if a feature provides or requires access, this specification can be dropped in this conversion since it is important only to define the topology of the architecture, which is already checked by the frontend. This covers all the mandatory definitions for a feature, additionally, it is possible to specify the data type of the feature. Since the data is a component, we need two information to identify it: the package and the type. While

subprogram and bus accesses target a subprogram or bus instead of a data component, the procedure is the same since they are identified by their type and package.

Listing 5.3 show the interface model of the component included in the system presented in Listing 5.1. It defines a single event data port exchanging messages with a specific data type. Given this definition the AAXML file can be extended as shown in Listing 5.4, the *features* tag is a child of the *component* tag, while this replicates the feature information in multiple places in the file, it also streamlines the code generation process in the next phase.

Listing 5.3: TODO caption

```
process custom_process
    features
        a_port: out event data port pkg::some_data;
end custom_process;
```

Listing 5.4: TODO caption

```
<component>
    [...]
    <features>
        <feature>
            <name> a_port </name>
            <direction> out </direction>
            <type> event_data </type>
            <datatype> some_data </datatype>
            <datatype_namespace> pkg </datatype_namespace>
            <category> K_PORT_SPEC_INSTANCE </category>
        <feature>
    <features>
</component>
```

CONNECTIONS – When dealing with connections we move outside the scope of a single component and we have to consider the interaction of multiple objects. For this reason, connections are defined, both in AADL and in AAXML, in the container, and they can connect two components in the same subcomponents set or a component and a frontier feature (i. e., a feature

defined on the frontier of the container component). First, there is a list of characteristics referring directly to the connection itself. The *name*, an unique identifier of the connection. The *kind*, a mapping in AAXML of the original Ada node kind, currently the only possible value is `K_CONNECTION_INSTANCE`, however, we decided to include it in the AAXML file to support future extensions. The *category*, as for features, connections may involve ports or accesses, this property specifies the type of the connection, it can be: `CT_PORT_CONNECTION`, for connection between two ports, `CT_ACCESS_DATA`, when connecting an access to a data component, or `CT_ACCESS_SUBPROGRAM`, when modelling a connection representing a remote subprogram call. Since connections are meant to describe the interaction between component, they carry information related to both components at the limits of the connection. Each description has a subsection called *port_info*, it includes the name of the source and the destination features, and the name and the type of the parent components.

Listing 5.5 shows the modelling of the implementation of a system where two processes are connected, one with an output port and the other with an input port. Listing 5.6 presents how this connection is mapped on the AAXML file, since the system is the root of the hierarchy, the *connections* tag is defined directly as a child of the root *system* tag.

Listing 5.5: TODO caption

```
system implementation root_system.impl
    subcomponents
        cmpA: process pkg::processA;
        cmpB: process processB.impl;
    connections
        con1: port cmpA.out -> cmpB.in;
end root_system.impl;
```

Listing 5.6: TODO caption

```
<system>
    [...]
    <connections>
    <connection>
        <name> con1 </name>
```

```
        <kind> K_CONNECTION_INSTANCE </kind>
        <category> CT_PORT_CONNECTION </category>
        <port_info>
            <source> out </source>
            <dest> in </dest>
            <parent_source> processA </parent_source>
            <parent_source_name> cmpA </parent_source_name>
            <parent_dest> processB.impl </parent_dest>
            <parent_dest_name> cmpB </parent_dest_name>
        </port_info>
    </connection>
    </connections>
</system>
```

PROPERTIES – In AADL, properties can be applied to every element in the model, moreover, in the definition of our meta-model, we introduced their importance in specifying components. Properties are such an important element of the languages, that additionally to all the already existing definitions, the designer can define his own set of new properties. Given their ubiquitousness, each AADL category has a set of general properties, plus specific ones, plus all the custom defined, the process of adding them to the AAXML file can happen at every point during the parsing of the tree. All properties defined in the model have at least two fields: the *name*, the unique identifier of the property as defined in the language definition or in the property set, and the *value*, the actual value of the property assigned by the designer in the specific instance of the model. Each property has a type (e. g., number, string, etc.), that is checked by the frontend for consistency but not replicated in the AAXML file, additionally, in AADL it is possible to define the measurement unit of a specific property, if present, it is included in the translated model using the *unit* tag. Default (e. g., *Period*) and custom (e. g., the *Default_name* of a topic) properties are managed in the same way, with the exception of the name of the property set containing the custom defined properties; it is included in the AAXML file using the *namespace* tag. In AADL it is possible to set properties at any point in the model, this means that in the property section of a component it is possible to reference every property of its internal parts (e. g., features, connection, subcomponents,etc.) or, through dot notation,

all the properties of the subcomponents. However, in AAXML, properties are defined in as direct children of the element owning them.

Listing 5.7 shows a definition of the implementation of a process component, it has a thread as a subcomponent and this thread has an output port on his frontier. As said before, it is possible, in the properties section of the process to specify both the properties of the subcomponent (i. e., *Period*) and of the feature (i. e., *Queue_size*). Listing 5.8 presents a portion of the AAXML file modelling the properties, while they were defined in the property section of the process, the *component* tag in the listing references the thread, since it is the owner of both the feature and the *Period* property.

Listing 5.7: TODO caption

```
process implementation componentA.impl
    [...]
    properties
        Queue_size => 1 applies to threadA.out;
        Period => 50 ms applies to threadA;
end componentA.impl;
```

Listing 5.8: TODO caption

```
<component>
    [...]
    <features><feature>
    <properties><property>
        <name> Queue_size </name>
        <value> 1 </value>
    </property></properties>
    </feature></features>
    <properties><property>
        <name> Period </name>
        <value> 50 </value>
        <unit> ms </unit>
    </property></properties>
</component>
```

FROM AAXML TO ROS/C++

The second step of the code generation is a model-to-text transformation. Differently from the first step where the original AADL model was converted in a different format, here multiple models (i. e., AADL and ANS.1 or JSON schema) are combined to create code artefacts. In particular, the output of the code generation after this step will be a single ROS package that contains all the custom messages, services and actions (if they exists), the source code of all the nodes modelled combined with any already existing problem-specific implementation, all the necessary build and configuration files, and the launch files matching the deployment configuration specified in the model.

The AAXML to C++ module is developed entirely in Python 3, exploiting the lxml XML toolkit [96], a Pythonic binding for the C libraries *libxml2* and *libxslt*, to parse and manipulate the XML-based input. This module is designed and developed based on a nested structure, similarly to a recursive approach, elements call each others in chain, and then the model-to-text transformation is executed from the most nested element to the root. More in details, each target ROS/C++ artefact (e. g., methods, classes, variables, etc.) has its own managing class in the Python implementation. Depending on the structure of the code to be generated, each of these classes contains all the necessary subclasses to define the correct output source code. Figure 5.4 shows a contained example following this approach, it presents the classes and their interactions to create the signature of a method. A C++ method signature is defined by the name, the list of parameter and the return type, the Python class managing the method contains the necessary code to output the name, but relies on other classes (i. e., *Type* and *Variable*) to generate the remaining components.

Differently from the first code generation step, the model-to-text does not output a single file, but it generates multiple artefacts. For this reason, the process is divided in multiple phases. The first phase is the set up of all the basic elements of the ROS package, with all the folder in place, it can create the launch files. Messages, services and actions are next in the list, when this phase is complete, the process looks for all the nodes and

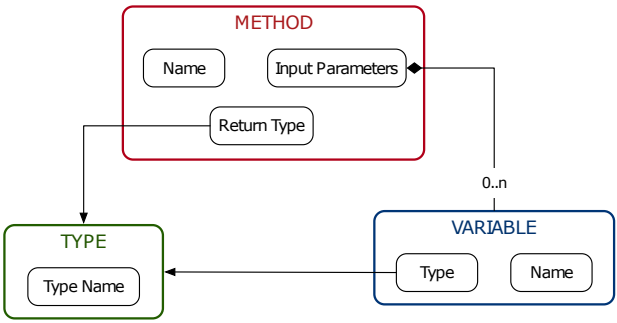


Figure 5.4: The classes, and their interactions, used by the code generator to manage a C++ method.

generates them. The last step is executed concurrently with the nodes generations, since it requires to incrementally fill all the configuration and build files.

PACKAGE DEFINITION – When implementing ROS nodes by hand, the first step is the creation of the containing package. The same process is followed by the code generator. By parsing the AAXML file, it can detect the definition of the AADL package and use it to create the corresponding ROS one. There is an important parallel between ROS and our AADL definition that makes the creation of packages easier; in ROS, a package is the minimal release unit, this means that all dependencies are package-based, this is true also for AADL, where elements can be defined in different packages and included at the beginning using the `with` keyword. In summary, from the name of the AADL package the code generator derives the name of the ROS package, and from the the `with` clause it creates all the necessary package-level dependencies.

Most of the package-creation process is the same for all packages. It defines the correct folder structure (see Section 3.1), and creates the *package.xml* and *CMakeLists.txt* files. These two files are initially created using all the dependencies information derived from the `with` clause, and they are later updated incrementally depending on the existence of custom messages and the number of custom nodes.

LAUNCH FILES – The second set of artefacts created by the model-to-text process are the launch files. In order to create a launch file, it is not necessary for the nodes to exist, for each process only the type (i. e., the compile-time name of the node), the package and the instance name are necessary to setup the structure. All this information is available in the AAXML file before generating any source file.

In our model, we defined a relationship between the hierarchical topology defined by AADL systems, and the structure of launch files. Therefore, the code generator parse the AAXML file looking for nested systems, and each one is converted in a launch file. Starting from the deepest child, the launch files are filled with all the necessary nodes, using the AADL instance name as runtime name, the AADL type as reference type, and, if present, the AADL package as the ROS package. To avoid unnecessary clutter in the architecture, only the connected processes are included in the launch files. Of course this is not limited to connections representing topics, but includes any kind of virtual or physical connection that justifies the role of the component in the system.

The following step of the launch file creation is the remapping of topics. In our model we distinguish between two different naming conventions both defined using properties. One is a port property, it is used during the generation of the source code and specifies the default (i. e., written in the source code) name of the topic. The other is a connection property, it is used when creating launch files to define the runtime name of the communication channel.

Last step is the assignment of parameters. If the designer used AADL properties to specify a configuration file, it is converted in YAML and assigned to the correct node in the launch files.

MESSAGES – The next phase of the model-to-text process is the generation of messages. Message types are modelled in AADL by a single data component augmented with a property that specifies its internal structure using ASN.1 or JSON Schema. Data types related to messages are not used in the model as instances, but only to specify the type of a communication, hence they only exist

in the package definition. This "out-of-system" use makes the code generation process simpler.

The package is analysed to detect all the data component; if they are associated with a message description, the file specified in the property is parsed by the specific code generator (i. e., ASN.1 or JSON Schema) to create in the `msg` folder of the package all the necessary ROS messages files. While creating the message files, the code generator also updates *package.xml* and *CMakeLists.txt* to include all the necessary messages dependencies and the modules to generate and build ROS message files.

NODES – The last step of the model-to-text process is the generation of the source code of the nodes. The complexity of this step is significantly reduced by our design of the meta-model. We defined a node in AADL as a process that uses threads as subcomponents evoking specific ROS functionalities. In particular, the *main_thread* characterises a process as a ROS node, this means that during the automatic code generation process the code generator can easily differentiate between ROS nodes and any other process by identifying the *main_thread* as a subcomponent.

The code generator analyses the package for any process definition including the *main_thread* as a subcomponent. For each one starts the process of creating all the necessary source files. Most of the basic C++ code is already defined in the engineered node superclass, therefore it is included directly in the automatic generated node. What the code generator needs to do, is to identify all the potential inner functionalities (i. e., publishers, subscribers and timers) of the node, to do so it uses a AADL thread-based approach. Currently the AAXML to ROS/C++ code generator is implemented to identify five different threads corresponding to specific designs:

- `ros::publisher.impl`. It correspond to the *source* component behaviour. In our engineered ROS node, it is implemented by a periodic timer that calls a publisher at the end. Given this design, the code generator has to generate both the timer with its callback and the declaration of the publisher.

- `ros::callback.impl`. It represent the *sink* component behaviour. In this case there is no special design, as in any other ROS node, it is a subscriber that trigger a callback. The code generator needs to declare the subscriber, register it to the correct topic, define the callback method and bind it.
- `ros::call_pub.impl`. This thread combines the previous two to evoke a *filter* component behaviour. Since it is a combination of a subscriber that call a publisher in its callback, the code generator needs to define and generate all the previous elements.
- `ros::service_provider.impl`. This thread is used to define a *reactive* component behaviour. The code generator needs to declare a service, advertise correctly and then declare the method used as callback.
- `ros::timer.impl`. This last thread does not correspond to any component behaviour, since it is not use for external interactions. However, ROS timers are useful to define the internal behaviour of the node, thus they are one of the potential thread identified by the code generator. Timers are defined similarly to the other elements, and trigger a callback when they expire.

If a component contains a thread of an unknown type, the code generation process continues by ignoring it. Often, the designer needs to include threads implementing special interfaces, for example a device driver may use a custom thread to interface with low-level hardware. Since it is impossible to predict all the possible configurations of a node, we decide to use an approach where the designer still has the flexibility of modelling a complete component, while the code generator will only perform a model-to-text transformation for known designs.

The code generator will create multiple files for the same node. All the implementation relative to the functionalities evoked by the threads is in the main source file of the node placed in the `src` folder of the package. To complete the generation of the main source file, it is necessary to add two more elements: the decoupled functionalities and the parameter initialisation process.

In both our model and in the generated code, we try to push the decoupling between the domain-specific implementation and the middleware-related code. In the model we achieve this by specifying subprograms as subcomponents of threads and then using the *Source_Text* properties to include the specific implementation of the subprogram. The code generation will retrieve this implementation and call it in the timer or subscriber callbacks as an external function. However, if the property is not specified, or the file referenced by it does not exist, the code generator will automatically create a corresponding header file that the developer can fill at a later stage. The inclusion of any external function or library is propagated by the code generator to the *CMakeLists.txt* file to maintain the package compilation-ready.

The last step of the node generation process is the definition of the internal state. This is done in a separate file that contains the *parameters* and *variables* structures as defined using ANS.1 or JSON Schema. Since they are basic types, parameters are completely defined and initialised using the default values provided, moreover, they are propagated in the main source filed to setup their interface to the parameters server. Variables are declared in their own structure, and initialised if they have a default value and a basic type. However, if they are a complex variables (see Section 4.5), they are only declared and their complex type is included in the header and propagated in the *CMakeLists.txt* file.

5.5 A COMPLETE EXAMPLE

Now that we have completely defined how to model a ROS architecture with a combination of AADL and a data modelling language, and we presented our approach to automatic code generation, we can show a complete example, going from the model to the source code. In this section, we will model a basic architecture composed by two nodes interacting to each other, and show the structure of the output package generated by the automatic programming toolchain.

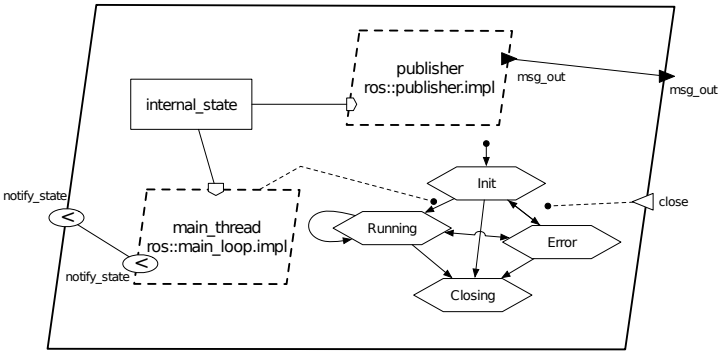


Figure 5.5: Graphical representation of the AADL description modelling a simple talker node implementing a publisher.

The micro architecture in question is the first introductory tutorial provided by the ROS wiki[3], slightly modified to include a custom message: a talker node implementing a publisher and a listener node implementing a subscriber connected through a topic and exchanging messages. Figure 5.5 show the graphical representation of the model of the *talker* ROS node. It evokes a *source* component behaviour by generating messages and publishing them on a topic. Additionally to the recurring subcomponents identifying our engineered ROS node (i. e., state machine, main thread and internal state), it models a publisher thread of type `ros::publisher.impl` communicating with the external environment using a data port specified by the message type. Not visible from the graphical representation are the properties of the node, they are shown in Listing 5.9.

Listing 5.9: TODO

```
Period => 10 ms applies to publisher;
topic_properties::Default_Name => "/out_chat" applies to msg_out;
Source_Text => ("talker.schema.json") applies to internal_state;
Source_Text => ("talker.h") applies to publisher.function;
Source_Name => "talk" applies to publisher.function;
```

3 http://wiki.ros.org/ROS/Tutorials

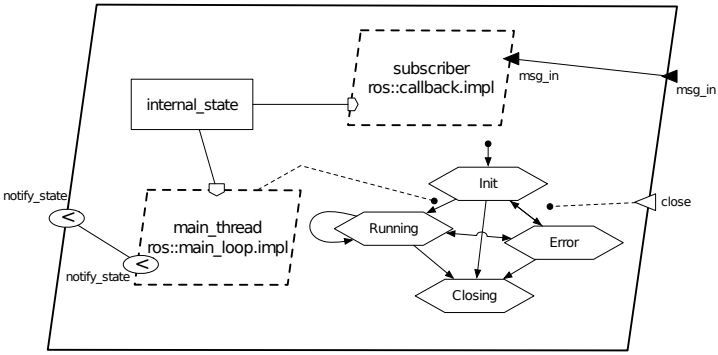


Figure 5.6: Graphical representation of the AADL description modelling a simple listener node implementing a subscriber.

Period defines the frequency of the publication, the thread will generate a message every 10 ms, this is used by the code generator to define the period of the corresponding ROS timer. *topic_properties::Default_Name* represent the name assigned to the topic during implementation when declaring the publisher. There are two *Source_Text* properties, one for the internal state, to define the structure of parameters and variables, and one points to the header included by the code generator to define the specific functionality of the publisher. This last property is combined with *Source_Name* to identify the specific function that will output the message to be generated.

Figure 5.6 presents the *listener* ROS node graphical AADL model. Since it receives the messages produced by the *talker*, it implements a *sink* component behaviour. As before, the model is composed by the elements characterising the engineered ROS node plus a subscriber thread of type `ros::callback.impl` receiving data from outside the component through an event data port that triggers the execution of the thread and it is specialised by a specific data component. As for the *talker* node, properties are used to refine the definition of the process, they are detailed in Listing 5.10

Listing 5.10: TODO

```
Queue_Size => 1 applies to subscriber.msg;
topic_properties::Default_Name => "/in_chat" applies to msg_in;
Source_Text => ("listener.schema.json") applies to internal_state;
Source_Text => ("listener.h") applies to subscriber.function;
Source_Name => "listen" applies to subscriber.function;
```

The properties of *listener* and *talker* are very similar. *Source_Text* and *Source_Name* have the same role: defining the internal state of the component and specifying the function to be called during the callback, in this case. Since the subscriber reacts to the messages received, the *Period* property is replaced by *Queue_Size* that specify the size of the message queue, in this case only the newest message is available. As before, there is a property that defines the name of the topic at compile time, this is the reference name used by the code generator when it creates the definition of the subscriber. It is important to note that the default topic name is different in the two nodes.

Listing 5.11 shows the missing element necessary to complete the architecture. At the beginning, the definition of the data component representing the ROS message. The *ChitChat* component is only an interface, its internal structure is defined through a property that points to the JSON schema file used by the code generator to create the ROS message. Follows the implementation of the container, since this is a small architecture with only two components, the root is also the only system. Here the two process instances are declared as subcomponent of the *talking.ros* system, moreover their topic communication is defined using a connection going from the output port of the *talker* to the input port of the *listener*. The properties of the system are used to define the specific runtime configuration of the architecture. *topic_properties::Name* applied to the connection specifies the remapping of the topic from the original default name used in the implementations. The two *Source_Text* properties applied directly to the processes are used to define a JSON file that contains a node configuration matching the schema applied in the definition of the components.

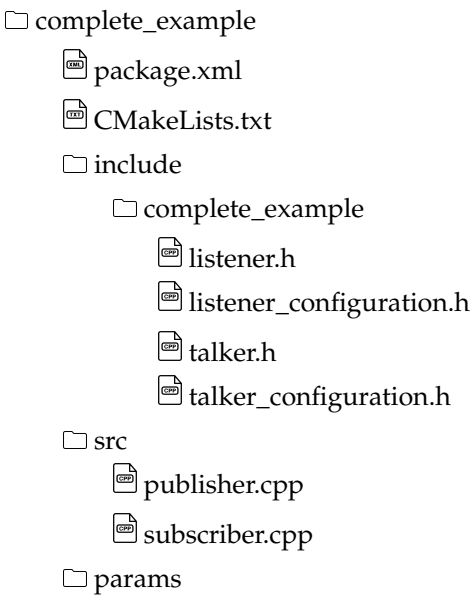
Listing 5.11: TODO

```
data ChitChat
    properties
```

```
        Source_Text => ("ChitChat.schema.json");
    end ChitChat;

system implementation talking.ros
    subcomponents
        talker: process talker.impl;
        listener: process listener.impl;
    connections
        chatter: port talker.msg_out -> listener.msg_in;
    properties
        topic_properties::Name => "/chatter" applies to chatter;
        Source_Text => ("talker.json") applies to talker;
        Source_Text => ("listener.json") applies to listener;
end talking.ros;
```

The AADL model, combined with all the necessary JSON and JSON schema files, is then parsed by the automatic programming toolchain. Since our model included also the source files containing the implementation of the functionalities of the nodes (i. e., *talker.h* and *listener.h*), the output of the process is a complete package ready for compilation. The structure of the package is the following:




listener.yaml


talker.yaml


launch


talker.ros.launch


msg


ChitChat.msg

6 ABSTRACTING THE ROBOT

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.

— John V. Guttag

In the previous chapters we discussed extensively how to capture all the details of a robotic architecture, how to codify not only the topology of the components, but also their inner functionalities. We defined a collection of tools to help the *system designer* and the *component developer* in describing, designing, realising and implementing their vision. However, this is only the starting point of a process to modernise the development of software for robots, to make the creation of robotic applications as accessible as developing a mobile app, a web page or a video game.

The shortest path to make a technology more accessible is through abstraction. Robot software development has already benefited from this approach with the introduction of robotic middleware and frameworks, for example, the ROS computational graph creates an abstraction layer between the underlying hardware and the components. This made robotics more accessible and triggered a process that resulted in vast repositories of components efficiently implementing basic robot functionalities (e. g., navigation, perception, manipulation).

However, the key element of abstraction is the context. Not every approach is useful to achieve a specific result. Technologies like ROS or other robotic middleware and frameworks are useful to streamline the development of components, but a different level of abstraction is needed to implement application at an higher level. Because of the success of component-based approaches, robotic system are becoming more complex and richer in functionalities attracting experts with diverse backgrounds that are more

interested in the high-level capabilities expressed by the robot as as system, instead of focusing on the low-level functionalities implemented by each component.

These *application developers* need a different type of abstraction that goes beyond the one provided by middleware and frameworks. In this chapter we tackle the problem of analysing the robot a system to capture the significant high-level functionalities and to create an abstraction that can be exploited to develop complex applications. First by defining an ontology that captures the structure of robot system, then using this framework to identify robot capabilities. These capabilities are the access point for creating an abstraction layer between a set of APIs and the underlying robot system. Lastly, we give an overview on how the model-based approach presented before can be used as a support for the concept of robot capabilities.

Contents

6.1 ONTOLOGY REPRESENTATION

Creating new abstractions layers on top of an existing technology is not a difficult task, multiple approaches can be used: programming interfaces, libraries, frameworks, middleware, domain-specific languages, and more. For example programming interfaces usually remain in the same context of the abstracted technology and only provides controlled access to the underlying functionalities, their aim is to hide the complexity of the implementation, not to create a decoupling layer. Frameworks push the abstraction a bit further, not only they hide the inner functionalities of the system but they provide entry points that a developer can use to extend the framework functionalities and create complex applications exploiting it. Domain-specific languages represent a more agnostic form of abstraction, since they present a completely different interface with respect to the target technology, however, by definition, they are created with a narrow scope. In conclusion, most of these solutions have the downside of being focused vertically (e. g., programming interfaces and libraries) or horizontally (e. g., middleware and domain specific languages), since the abstraction they provide is usually created to target a specific category of user.

In an effort to achieve a more generalised abstraction from the underlying robot system, we decide to base our approach on an ontology. Ontologies have been successfully used to limit the complexity of the domain and to organise the information into data and knowledge. The advantage of ontologies, with respect of the other methods mentioned before, is that they provide a strong technological decoupling between the original domain and the resulting classification. Ontologies are an universal language that is not bound to physical platforms, implementations, or design choices. All these characteristics makes an ontology the perfect tool to create an abstraction layer between the robot as a whole and a potential *application developer* or any other external actor designed to interact with it.

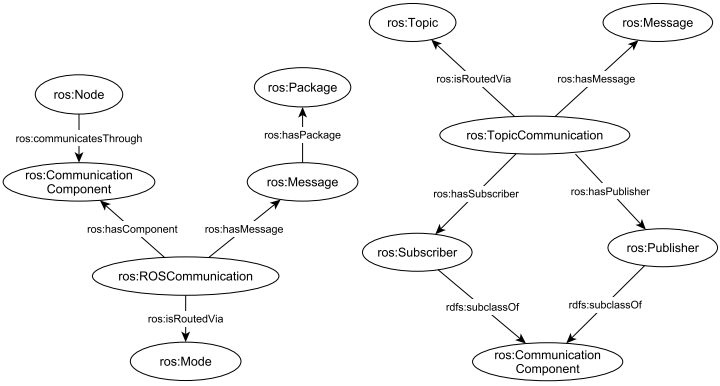
What we want to achieve with our abstraction is to extract from a running robot a list of capabilities. Since our focus is on decoupling the robot from the high-level applications, in this work,

we will not impose any specific definition of capability. In this context, a capability is anything, at any level of complexity, that the robot can do. For example, any form of movement or any form of sensing are capabilities. A robot arm with three joints will have three capabilities, each one associated one joint, plus one evoked by the movement of the entire kinematic chain. We adopted an ontology instead of other approaches also to provide flexibility to our classification, in an ontological description it is not difficult to implement horizontal or vertical relations and to create subclasses and specialisation of the existing entities.

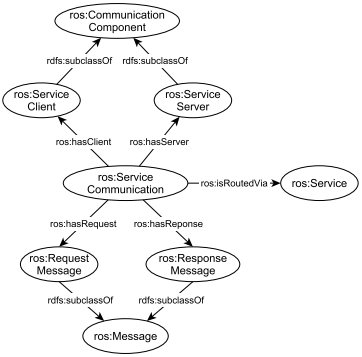
ROS DESCRIPTION

Before the definition of robot capabilities it is necessary to create a formal representation described as an ontology of the main elements of ROS. While this is not used directly in the capability extraction process, it creates a structure that can be used as a reference and to contextualise the capabilities. Since the aim is not the creation of an ontological classification of ROS, but to define a framework for capabilities classification and extraction, we did not capture in the ontology every element of ROS.

The final aim is to automatically extract which capabilities are exposed by a specific system, therefore, we can only exploit static information provided by the source code or dynamic information obtained from the runtime of the system. Given that ROS support various runtime-only configuration (e. g., topic remapping, parameters definition at start-up, dynamic reconfigure, etc.), we decided to focus on all the information obtainable when the system is running. In ROS, at execution, the main source of information comes from the runtime instance of the Computation graph, a collection of all the active nodes and their connections. Various elements, other than nodes, are part of the graph, for example the ROS master or the parameters server, to build our ontology, we decided to focus only on those parts that are defined by the developer, both as a topology definition and as a complete implementation. In particular, we take in account:



(a) TODO **(b)** TODO



(c) TODO

Figure 6.1: TODO

- A *node* is a process performing a specific computation and represent the minimal executable unit of ROS. A single node or a collection of nodes implement a specific functionality, therefore is connected to a potential capability of the robot.
- *Messages* are the typed data structure exchanged by the nodes when communicating with each other. Here we refer to messages in the broader sense, including all communication object exchanged in any type of communication. Messages are often generic (e. g., *Vector3* in the *geometry_msgs* package), but when specific (e. g., *Image* in the package *sensor_msgs*) they can be used to identify capabilities.
- *Topics* are the named channels used to exchange messages asynchronously using a publish/subscribe paradigm. Since they are completely user-defined, it is not possible to infer any significant information from them, however, they are fundamental to categorise capability, especially if multiple instances of the same one are available (e. g., multiple vision systems).
- Similarly to topics, *services* are named channels for synchronous communication, exchanging pairs of messages (i. e., request and response). Again, the service itself does not provide significant information since it is user-defined, however, it helps categorise the identified capabilities.

While they are an alternative communication system between nodes, actions are not included in this analysis for multiple reasons. First of all, they are rarely used, therefore any extra effort to include them would provide little or no extra information. Additionally, given their complexity, they are, in most cases, completely defined by the developer and not based on any pre-defined message. Lastly, from a pure communication point of view, actions are only a collection of topics with a specific set of rules hidden to the developer by the action clients.

Focusing on these elements we can define a formal representation of ROS and the interactions of its components. As said before, the aim of this ontology is to support the definition of the concept of capabilities and their automatic extraction from

an existing system, therefore, while it is based on the Event and Situation ontology design patterns, it is not rigorously designed for robustness, completeness or originality. Figure 6.1 show a graphical representation of the ontology describing the main ROS elements (Figure 6.1a), and a more detailed description of topics (Figure 6.1b) and services (Figure 6.1c).

BASIC ROS DESCRIPTION – Figure 6.1a capture the main elements of ROS and how they are conceptually interconnected. ROS revolves around the concept of the Computation graph, where nodes (i. e., executable components) are connected to each other through different type of communications (i. e., topics and services). Messages are the main carrier of information between different components. In ROS, there is one way to loosely aggregate elements by functionalities: packages. Nodes and messages have an assigned package, which is user defined, therefore not all of them are a reliable source of information to define the expected functionalities of an element. Nevertheless, some effort to create standard packages exists, some examples are *sensor_msgs* for messages used to encode sensor measurements, or *navigation* for the planning, mapping and control nodes.

All these details are codified by the ontology. *ros:Node* and *ros:Message* are defined as part of a specific *ros:Package*. The edges of the runtime graph are identified by the *ros:ROSCommunication* class, multiple relations exists between this class and the rest of the ontology. To identify the type of communication, we use the *ros:Mode* class and the relation *ros:isRoutedVia*, together they specify if a pair of nodes use an asynchronous (i. e., topic-based) or synchronous (i. e., service-based) communication system. The relation between *ros:Node* and *ros:ROSCommunication* is not direct, but it is mediated by a *ros:Communication Component*. This class represent the interface implemented by the node to use a specific communication pattern, moreover, it is important to define the direction of the communication. For example, the a velocity message published by a single node and read by a multitude is more probably related by a speedometer, while multiple velocity publisher converging on a single subscriber can point to a set-point multiplexer.

This ontology was defined independently from the modelling approach defined in Chapter 4 and with no intention of establishing a encompassing definition. Nevertheless, there are recurring structures that appear both in the ontology and the model. Clearly some are related to the internal structure of robotic middleware and frameworks, or mode in details to ROS itself. For example the definition of nodes (i. e., robotic components in the *component-and-connector paradigm*) and messages (i. e., communication objects) appears in most component-based approaches. However, the necessity of specify a communication component acting as an interface between the the node and the communication system, is the same as the need to define AADL threads as a way to model the inner functioning of a component. In summary, it is never enough to just specify a components and its connection, it is necessary to detail the communication, its direction, the protocol, and the management system.

TOPIC DESCRIPTION – Figure 6.1b represents the extension of the previous ontology to better describe communication happening through topics. The central class of this ontology is *ros:TopicCommunication*, which is a subclass of the more general *ros:ROSCommunication*. Here we specify more in detail the protocol used by this type of communication by creating a subclass of *ros:Mode* and defining *ros:Topic*. The topic-based communication is the simplest protocol provided by ROS, implementing an asynchronous interaction based on the publish/subscribe protocol, additionally it uses a publish-and-forget approach. This is translated in the ontology by defining two subclasses for the communication component: *ros:Subscriber*, as the subscriber of the protocol, in charge of receiving the messages, and *ros:Publisher*, as the publisher, in charge of generating the messages. The lack of ownership of the messages and the absence of delivery confirmation result in a simple relation between the message and the topic.

To understand how an instance of a topic is defined with respect to the corresponding nodes, we can take as an example a simple pair of nodes implementing a local planner feeding a control system with velocity set-points. They are connected by a named topic (`/cmd_vel`) and exchange a specific type of message

(*geometry_msgs/Twist*). The local planner implements a publisher, while the control system implements a subscriber. The resulting representation is in Listing 6.1.

Listing 6.1: TODO

```
@prefix ros: <onto-ros/class#>
@prefix : <onto-ros/resource/>
:setpoints a ros:TopicCommunication;
    ros:isRoutedVia :cmd_vel;
    ros:hasMessage :twist;
    ros:hasPublisher :cmd-output;
    ros:hasSubscriber :cmd-input.
:twist a ros:Message.
:cmd-vel a ros:Topic.
:local-planner a ros:Node;
    ros:hasComponent :cmd-output.
:controller a ros:Node;
    ros:hasComponent :cmd-input.
```

SERVICE DESCRIPTION – Figure 6.1c represent a second extension of the basic ontology to describe the synchronous communication approach provided by services. As for topics, the central class is *ros:ServiceCommunication* as a subclass of *ros:ROSCommunication*. To identify the protocol implemented by services, we created a dedicated subclass of *ros:Mode* called *ros:Service*. Given their synchronous approach, messages exchanged during the execution of a service are divided in two categories: requests, sent by the client to trigger the functionality provided by the service, and response, sent by the server with the result of the computation or as a completion notification. To capture this we defined two subclasses of the original *ros:Message*: *ros:Request Message* used to classify the request sent by the client, and *ros:Response Message* used to identify the response sent by the server. As for the topic, it is necessary to specify, using subclasses of *ros:Communication Component*, the specific interfaces implemented by the node when acting as a client or as a server. The two subclasses are: *ros:Service Client*, for the client, and *ros:Service Server*, for the server.

Listing 6.2 provides a small example on how an instance of a service is represented using the ontology. In the example there is a global planner that needs to synchronously request an oc-

cupancy grid from the map server. They interact using a named channel (`/map`), the client (i. e., the global planner) send an empty message (*std_msgs/Empty*) as a request to the server (i. e., the map server) to trigger the delivery of the map through the response (*nav_msgs/OccupancyGrid*). Each node, depending on its role, implements a service client or a service server.

Listing 6.2: TODO

```
@prefix ros: <onto-ros/class#>
@prefix : <onto-ros/resource/>
:map-service a ros:ServiceCommunication;
      ros:isRoutedVia :map;
      ros:hasMessage :occupancy-grid;
      ros:hasClient :map-request;
      ros:hasServer :map-response.
:occupancy-grid a ros:Message.
:map a ros:Service.
:global-planner a ros:Node;
      ros:hasComponent :map-request.
:map-server a ros:Node;
      ros:hasComponent :map-response.
```

CAPABILITIES EXTRACTION

The ontology defined in Figure 6.1 is useful to understand the most significant and recurrent elements of a ROS-based system. The main insight is the recurring relationship between a node, a message and a specific communication pattern (i. e., service or topic). These three elements together can be used to identify high-level capabilities of the robot and potential entry points.

To better understand how this triple (i. e., node, message, and communication patter) can identify a capability, we can recall the examples described in the previous section. Let us start by analysing the pair nodes involved in a simple control subsystem. On one side, there is the local planner, it uses a publisher to generate a *Twist* message on the `/cmd_vel` topic. We cannot extract significant knowledge from the node itself, except for the fact that it implements a publisher, therefore we can infer the direction of the communication. Since *Twist* is one of the standardised

messages, we know that is commonly used to exchange set-points (speed sensor usually rely on *Odometry*), hence we can conclude that this specific message published by the node on a specific topic can be read to have an insight on the expected velocity of the robot. On the opposite side of the topic there is the control node. Since they are connected directly, it shares the same type of message (i. e., *Twist*) and the same topic (i. e., `/cmd_vel`) of the local planner. However, the control node implements a subscriber, this changes completely the meaning of the message. While the assumption that *Twist* is used for set-points still stands, the different direction of the communication (i. e., from the topic to the node) let us know that we can use this message on this topic to control the movement of the robot.

A similar analysis can be done for the second example involving services. This pair of nodes implements a global planner requesting a map from the map server. On the provider side there is the map server, it implements a service server and advertises it with the name `/map`. As before, the named channel is completely user-defined, therefore there is no knowledge we can extract from it. From the node, since it implements the server, we can infer the direction of the communication: the map server receive the request and provide the response. On the receiver side there is the global planner, it accesses the service advertised by the map server using a service client. By this communication interface we can infer that the global planner will start the communication exchange. In the synchronous protocol of the service, two message are involved in the communication, in this case the request is an *Empty* message, and the response is an *OccupancyGrid* message. Since they are standardised messages, we already know to which functionalities they are usually related. An *Empty* message carries no information, hence it can only be used as a trigger to activate another functionality. *OccupancyGrid* is the standard message to share a bidimensional cell-based map where each cell can have three possible values (i. e., free space, obstacle or unknown), therefore we can easily assume that the result of this message will be a map of the environment. In summary, we can infer that both nodes support a map representation, the map server as the provider, hence we can trigger the service to receive a map, and the global

planner as the receiver, this means we can setup a compatible service to impose a map to the planner.

By analysing these two examples, we can infer that establishing the mapping between a ROS-based system and the capabilities offered by the corresponding robot is equivalent to establishing a relation between a capability and a triple created by a node interface, a message and a communication protocol. In practice, there are four types of triples that we can define, that are a specialization of `<ros:Communication Component, ros:Message, ros:Mode>`:

- `<ros:Publisher, ros:Message, ros:Topic>`
  This triple represents a capability evoked by an asynchronous communication. Since it is a publisher, it is possible to exploit it to interact with the system (e. g., teleoperation).

- `<ros:Subscriber, ros:Message, ros:Topic>`
  Similar to the one before, this capability is tied to an asynchronous channel. However, since it is evoked by a subscriber, it can be used to receive information from the system (e. g., sensing).

- `<ros:Client, ros:Request, ros:Service>`
  This triple represents the relation between a client and the synchronous protocol. By abstracting the interface created by this communication, we can control the expected response and inject a specific data content in the system.

- `<ros:Server, ros:Response, ros:Service>`
  This is the opposite side of the service-based communication protocol. By knowing the interface of the server, we can trigger its functionalities independently from the architecture.

As said before, the final aim of this approach is to create a system to automatically extract the capabilities of an existing architecture. As a result, it is necessary to define strict and clear rules relating a specific element of the ROS system to a capability; unfortunately, most of the architectural components of ROS are defined by the developer. There is no direct relationship between functionalities and node implementing them, for example, *move_base* incorporates a global and local planner in the same node, but a

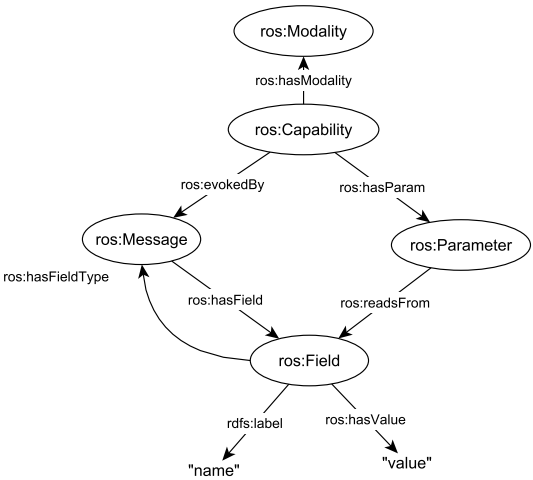


Figure 6.2: TODO

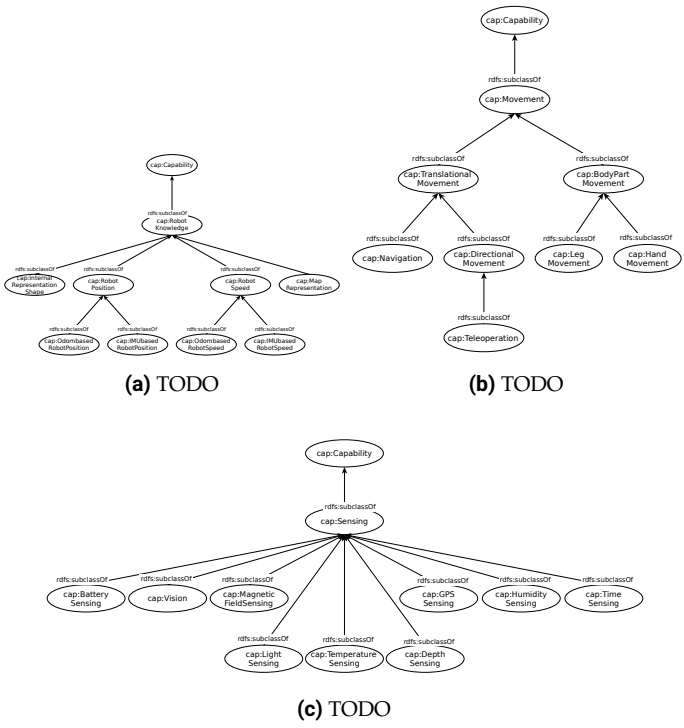
developer could implement them separately. Moreover, topics are completely user defined, they can be renamed at deployment time, or duplicated, therefore there is no connection between the name of the topic and its functionality. Lastly, the use of messages is unrestricted, a very pedantic developer could decide to use only custom messages in his architecture. Thankfully, ROS provides a wide variety of already defined messages, and, through component reuse, a self-standardisation process emerged. For instance, a *Pose* message from the *geometry_msgs* package will, most of the time, provide information about the pose of the robot in a tridimensional space.

Given all these considerations, the resulting ontology to relate capabilities with portion of a given ROS architecture is presented in Figure 6.2. On the left hand side of the figure, we recall the three main elements related to a specific capability: message, communication component and mode. Since the only semi-standardised element of this triple is the message, a capability is evoked directly by a specific message. For instance, a *LaserScan* message evokes the capability of sensing the environment using a laser sensor. Capabilities are characterised by a modality, they can be *read* or

write. A capability has a *read* modality when it can provides information to an external user, for example, collecting the map from the map server is a *read* capability. The *write* modality means an external user can influence the behaviour of the architecture, for example, teleoperation is a *write* capability. The modality of a capability is defined by the type of communication component associated with it: publishers and servers are associated with *read* capabilities, since an external user can collect information by reading their output, subscribers and clients evoke *write* capabilities, it is possible to change the behaviour of a system by providing a specific input. In some cases capabilities express both *read* and *write* modalities. An example are all the sensing-related capabilities, an external user can read directly the output of a sensor or act as a sensor to change the behaviour of the system. In other cases only one modality is available, for instance, teleoperation is *write*-only, because reading the joypad input is more related to sensing. The last part of the ontology is related to the application of the capabilities, the objective is not to just identify them, but to use them as an entry point for interacting with the robot. For this reason, each capability has a set of parameters, which are directly related to the fields of a message. A parameter can read from (or write to) a specific message field to create an instance of a message to interact with the robot architecture.

CAPABILITIES TAXONOMY

In the previous sections, we defined the ontological superstructure necessary to formalise a ROS-based system (i. e., Figure 6.1), to define the relation between ROS elements and capabilities (i. e., Figure 6.2), and to extract capabilities from a running system (i. e., binding between messages and capabilities). This approach is completely general and not bound to any specific definition of the concept of capability other than *"something the robot is capable of"*, it can be as high-level as *"navigate to a goal"* or as low-level as *"last fix of the GPS"*. However, to be able to apply this approach to a real robotic system, we defined a potential taxonomy of capabilities; the result is presented in Figure 6.3.



(a) TODO **(b)** TODO

(c) TODO

Figure 6.3: TODO

The root of the taxonomy is a general unspecified capability. From this there are three possible subclasses: sensing, robot knowledge, and acting. These three subclasses are meant to recall the *sense-plan-act* paradigm, and focus on the three main characteristics of a robot: the ability of understand the environment through sensor, the ability of process the inputs and represent them, and the ability to interact with the environment through actuators.

In the *cap:Sensing* subclass there are capabilities covering different type of sensing, both for perception (e. g., *cap:Vision*) and proprioception (e. g., *cap:Battery*). In this taxonomy, the sensing capabilities are directly connected to a specific sensor (e. g., a magnetometer for *cap:Magnetic Field*), however, under this macro-

category it is possible to fit capabilities obtained after one or multiple processing steps, if they provide direct information about the environment. For instance, AR tags detection could be a *cap:Sensing* capability, since it provides the position of a particular object in space after processing an image. It is difficult to define the frontier where an input has received enough processing to lose the status of sensing, an option is to consider the need of external knowledge. A GPS driver process an input stream using a standardised process, therefore is sensing, however, estimating the position of the robot from a velocity measurement requires specific information about the kinematic of the robot, hence it cannot be sensing.

In the *cap:Robot Knowledge* subclass fall all those capabilities that are related to any internal representation used by the robot (e. g., *cap:Map representation*), or any intermediate processing done to achieve the interaction with the environment (e. g., *cap:Gobal planning*). This is potentially the largest subclass, since it encompass anything in between the sensors and the actuators, this is done on purpose, since it is difficult to categorise the functionality of a robot. In our version of the taxonomy, we mostly focus on mobile navigation, by including capabilities related to the robot position, the mapping of the environment and the physical shape of the robot. Most of these capabilities may not be relevant for a different type of robot, for example a manipulator may include object classification, grasping configuration estimation or inverse kinematic.

The *cap:Acting* subclass may be the simplest to define, any capability that cause an action of the robot falls here. Any direct or indirect interaction with the environment, or change of configuration of the robot can be considered part of the *cap:Acting* subclass. Again, we focus mostly on mobile navigation, therefore our taxonomy includes movement-related capabilities, like *cap:Navigation* or *cap:Teleoperation*. However, any other form of interaction can be categorized in this subclass, for instance, speech, grasping or visualisation (e. g., a robot equipped with an onboard tablet).

6.2 ROBOT APIS

A formal definition of robot capabilities (functionalities, actions, tasks, etc.) is a topic often discussed, but very difficult to frame and resolve. We do not have the audacity of suggesting that our approach is the only possible solution, and more importantly, we know for sure that the taxonomy we presented is far from being exhaustive. However, our aim was never to provide the definitive classification for robot capabilities, it was a more practical one: to create an abstraction layer between the functionalities provided by modern robotic middleware and a potential *application developer*. What we want to achieve is to view the robot architecture as a whole, a system providing a set of entry points that a developer can use to create high-level applications exploiting the capabilities of the robot.

The motivating scenario that pushed us in the direction of considering robotic platforms as abstracted entities providing a common interface is the integration of robots in smart-cities. Milton Keynes is a city in Buckinghamshire, England, that engaged in a series of programs to develop a "Future City". One of this was the MK:Smart project, which has developed a state-of-the-art data acquisition and management infrastructure (the MK Data Hub) and an IoT network of sensors. The MK Data Hub was built with the idea that a common facility to efficiently manage, integrate and re-deliver data from a variety of sources could be exploited by applications and services, reducing their development costs and enabling intelligent data management (mining, analytics, aggregation, alignment, linking) at the scale of the entire city. Given this, it is just a short step to introduce robots as an additional actor in the network of sensors connected to the smart-city; to integrate them as as data collectors [90] and data consumers [30].

To better clarify the role of robots in the smart-city as actors of the integrated data acquisition and management infrastructure, we can provide a simple example. In Milton Keynes, a door-to-door robot-based delivery system is currently active. These robots travel around the cycle paths to deliver grocery upon request and are equipped with various sensors including cameras. At this moment, the information collected by the robot during their deliveries is

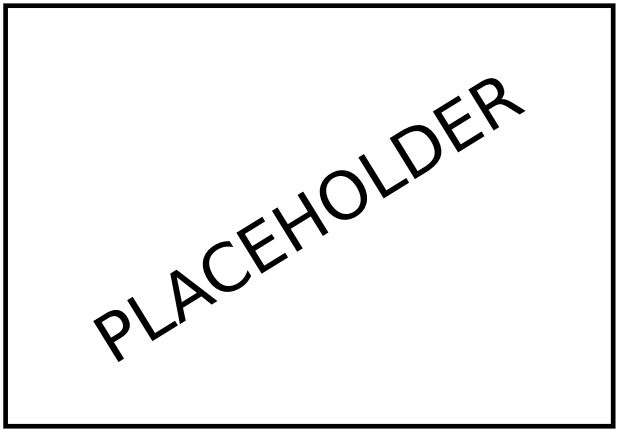


Figure 6.4: TODO

not used for any additional task, but by integrating them in a larger system they could use their cameras to spot abandoned garbage on the cycle path. Of course, the delivery robot itself cannot pick up the trash, but it can notify the centralised data system about its discovery, the system will then aggregate all the information and setup a path for a garbage collector robot to clean up the cycle path.

With the current technology landscape for robotics this application would require the development of a series of ad hoc interfaces between each of the robot involved and the centralised data management system. In the particular example presented there is the additional complication that the delivery robots are owned by a private company, which is not keen to disclose the internal functioning of its machines. All these problems can be resolved by introducing an abstraction layer.

In this section, we present all the necessary tools we developed to create a set of robot APIs based on our definition of capabilities that can be used to completely abstract a ROS-based architecture. Figure 6.4 shows an overview of the complete system. Some elements are ROS-bound and are used to setup the abstraction layer, others are independent from the architecture and define the agnostic APIs to interact with the robot.

ROS-BOUND INTERFACE

Figure 6.4 presents the complete set of elements necessary to interact with the robot. Three of them are enclosed in a container labelled *ROS*. The reason of this distinction is because those three elements require a ROS-enabled environment to work. The first one is, reasonably, the robot, the collection of all the software and hardware components necessary to implement the functionalities provided by the system. There is no required on the structure of the architecture to be compatible with our approach, any ROS-based robot can be used. The only caveat is that our process, as described in Section 6.1.2, is based on the assumption that some specific ROS messages are used as expected from their semantic (e. g., *Twist* is used for velocity). Directly connected to the robot, there is the *Analyzer* and the *Dynamic node*.

ANALYZER – Before being able to interact with the robot, it is necessary to identify which functionalities it implements. In this initial phase is where the capability ontology described in Section 6.1 is used. While defining the relation between capabilities and ROS elements we highlighted how a capability evoked and specified by three elements: communication interfaces, messages and communication channels. The role of the *Analyzer* is to use a list of binding between message types and capabilities to create a knowledge base containing all the available functionalities of a specific robotic platform.

Thanks to the effort spent in defining the ontology, the process of extracting the capabilities is quite straightforward. First of all, it is necessary to start the complete robot architecture, when all the nodes are up and running, we can run the *Analyzer*. The *Analyzer* loads a list of message/capability binding, then analyse the entire graph of the target ROS architecture. For each topic or service and their associated communication interface (i. e., publishers and subscribers for topic, client and servers for services), the *Analyzer* adds an entry in the knowledge base. Each entry create a relationship between a communication channel and a specific capability through a message. Recalling the example of the controller introduced in Section 6.1.1, the entry in the know-

ledge base would create a relation between the `/cmd_vel` topic, the *Directional Movement write*-capability and the *Twist* message.

A significant upside of creating a knowledge base instead of using a runtime approach is that the output of the *Analyzer* is permanent. If the architecture does not changer, there is no need to run the component again. This means that robots with the same architecture shares the same capability knowledge base, and that a single knowledge base can be created at deployment-time and shipped together with the architecture of the robot.

DYNAMIC NODE – Together with the definition of the capabilities, there is another element necessary to correctly implement an API system: a reliable interface with the ROS architecture. Given the structure of the ROS Computation graph (i. e., communication channels supporting an *n-to-n* cardinality), it is not difficult to inject or read messages from the architecture. In fact, ROS itself provides various tools to interact with the system at runtime (e. g., `rostopic` and `rosnode` family of commands). However, most of these commands are meant to be run from a terminal and designed for the occasional interaction, hence are not suitable to be used as a unified interface between the robot and an external system.

For these reasons, we implemented the *Dynamic node*; its role in the API system is as simple as it is crucial. It relays the interaction coming from capability-based environment of the external user to the ROS-enable environment of the robot by acting as a frontier between the two. The *Dynamic node* implements a system to create and destroy dynamically the suitable communication interfaces associated with a specific capability. To better clarify, let us take again the example of the velocity controller. From the *Analyzer* we know that the node exposes a subscriber to the topic `/cmd_vel` and the message exchanged is *Twist*. The *Dynamic node* can use these information to dynamically create a publisher on the specified topic with the specified messages, and then, upon request, publish content on the topic.

In order to make the approach more general, the *Dynamic node* does not require the content of the message to be defined as an actual ROS message. It expects a Python dictionary with the

same structure of the message. Moreover, it supports multiple form of communication: one-shot interactions with the topics (i. e., publish or read a single message), complete bidirectional relay of messages, and client and server replacement. The Dynamic node is developed for generality, while the superstructure provided by the capabilities makes the interaction easier, it can be used directly as a dynamic interface to the architecture. In summary, it provides a semi-agnostic (i. e., it requires a ROS-enabled environment to run and the knowledge about topics and messages) local API to interact with a ROS-based system.

ROS-INDEPENDENT INTERFACE

By going back to Figure 6.4, it is possible to see that no other element is inside the *ROS* container. This means that the rest of the system is completely ROS-independent and can be run on any platform. The way we structured the APIs is guided by the smart-city and robot interaction scenario. What we wanted to achieve was to define an interaction approach that a remote data management system can use to receive information from the robot and send commands and instruction without having to implement any ROS-related (or, more in general, robotic-related) interface. By following these specifications, we developer a *Server* and, finally, the *OntoRob Interface*. Of course this approach can altered by removing the *Server* and integrating some of its functionalities in the *OntoRob Interface*; the result is moving from a remote APIs approach to a local one.

SERVER – The role of the *Server* is to create a completely ROS-agnostic interface to the robot. Differently from the *Dynamic node*, not only the the structure of the messages is not related to ROS, but also there is the shift from the *node/message/topic* paradigm, to *modality/capability/parameter*. In summary, the *Server* is the capability-centric interface to the robot.

The *Server* provides a list of remote APIs that can be called to receive information or interact with the target system. Since it has access to the knowledge base it can provides the list of the active capability with their corresponding topic/service name. In

this case, the name is only an identifier, since the same capability can appear multiple times in a system (e. g., two *Vision* capabilities from stereo cameras), we need a way to identify them. As described in Section 6.1.2, each capability has a list of parameters that mirrors the fields of a ROS message. These information can be used to activate the `/read` and `/execute` APIs of the *Server*.

During his normal execution, the *Server* subscribes through the *Dynamic node* to all the topics associated with *read*-capabilities. When an external user triggers the `/read` API using a pair capability-topic, the *Server* will first check in the knowledge base that the requested pair is legal, and then it will answer with the last messages exchanged on the topic, suitably converted using the parameters of that specific capability. With the opposite approach, when an external user triggers the `/execute` API, the *Server* receive a triple composed by: a *write*-capability, a topic name, and the parameters corresponding to that capability. As before, first the *Server* checks for the command validity and then converts it to a suitable data structure to feed it to the *Dynamic node*. To increase the flexibility and the generality of the interaction, all the exchanges between the external user and the *Server* are done using JSON.

ONTOROB INTERFACE – In theory, the *Server* provides a complete enough interface to remotely interact with the robot. It gives all the necessary capabilities, it provides the output of topics or services, and receives commands. A centralised data management system like the MK Data Hub that triggered the implementation of this approach would directly interact with the *Server*. However, our long term vision was to develop a interface to support the development of robotic applications, hence we developed a Python-based that can be easily integrated in a computer program as a developer would do for any other API.

The *OntoRob Interface* is a class that can be instantiated to establish a connection with *Server* and hide the complexity of the JSON-based interaction. The class provides a more streamlined and compact interface that can be used to integrate the functionalities provided by the *Server* in a Python program. It has method to obtain knowledge about the current active robotic system: the list

of topic (as before, used as capability identifiers), the list of capabilities, the relation between the two, the structure of messages. Additionally to all these methods, it provides APIs to retrieve information from *read*-capabilities and interact with the robot through *write*-capabilities.

6.3 BRIDGE MODELS AND CAPABILITIES

At first glance, the connection between the abstraction provided by capabilities and a model-based approach seems quite loose. Surely both approaches have the same final aim: simplify and streamline the development of robot-related software, enhance robot architectures with support tools, and help the developer create, design and implement complex robotic applications. However, this can be said by quite a few approaches: frameworks, middleware, libraries, design patterns, collections of good practices. Hence the question: since they have a common goal, how these two systems can interact and benefit from each other?

In this chapter, we described how to extract the capabilities of a robot we had to create a binding between a capability and a ROS message. This approach allows us to analyse a running ROS architecture and infer which capabilities are evoked. While it was successful one the architectures we tested it, this strategy is not particularly robust, since there is no intrinsic structure in ROS messages. For instance, *Twist* is commonly used to define three linear velocities (i. e., v_x, v_y, v_z) and three angular velocities (i. e., ω_x, ω_y, ω_z), however, it is not uncommon to use ω_z to define the steering angle in an Ackermann kinematic. This is an use against the expected semantic that at least maintain the general capability, however, nothing stops a developer for using a simple *Vector3* to control a mobile platform, breaking the expected relation between the capability and the message.

A way to make the capability extraction process more robust is to use the model. Instead of relying only on the runtime graph, the designer can specify which capability is evoked by a particular component by using a property in the model. By having a ternary

relation between message, component and capability, we can verify the result of the automatic process. A capability evoked by a message is confirmed by the related component, moreover, this additional capabilities can be used to create a more complete interface with the underlying platform.

The beneficial relation between the two approaches is bidirectional. Not only the capability extraction system benefits from the existence of the model, but also the model-based design can be enhanced by the use of capabilities. With the current tools available to designers, it is possible to check the consistency of the model (e. g., the correct data is exchanged on a topic), but it is not possible to check if an architecture is functionality complete. By extending the capability taxonomy in an ontology, it is possible to define dependencies between capabilities. For instance, to achieve navigation, it is necessary to know the current position of the robot, perform global and local planning. In a capability-tagged model the dependencies ontology can be used to verify that a specific capability has all its dependencies satisfied. Moreover, by selecting before the list of target capabilities of the architecture, it is possible to check if the system is functionality complete and if it can fulfil all the expected task.

Tagging the model with capabilities is also useful to define a library of reusable designs. A designer could only specify a capability and receive a list of all the components and collections of components necessary to evoke that specific capability. This could be pushed even further, with a complete specification of the expected capabilities of a robot and a library of correctly tagged designs, it would be possible to create an automatic architecture builder system. The designer specifies all the necessary capabilities, the dependencies ontology fills the gaps, eventually with human intervention, then from the design library an automatic system selects all the necessary components. The designer just needs to connect the component together to finalise the architecture.

In summary, while the two approaches are aimed at two different categories of actor involved in the design and development of robotic architectures and applications, they are part of the same effort and can benefit one from another by creating a more com-

plete development stack going from the bare physical platform to the high-level applications.

7 EXPERIMENTAL EVALUATION

Conta ciò che si può contare, misura ciò che è misurabile e rendi misurabile ciò che non lo è.

— Galileo Galilei

Usually, robotics is a prime example of applied engineering. Everything starts with a practical problem, for instance, how to autonomously navigate in a crowded environment, from that, multiple technological solutions are imagined, designed and implemented. When a solution is ready to be executed is deployed on the robot and tested. A failure results in a step back, in creating a different solution, while a success pushes the engineer to look for a new problem.

Unfortunately, our work does not target a robotic problem, but a robotic meta-problem. How to make the process of creating technological solutions more streamlined and more accessible, in a way that failures are only minor setbacks in resolving a problem. This makes the experimental evaluation different and more difficult, since it has to include the human element.

In this chapter, we present two experimental evaluations that we used to test and asses our approaches. First, we present a robotic use case, where the architecture of an autonomous wheelchair is implemented from scratch using a model-based approach combined with automatic code generation. Then, we push the boundaries of our capabilities-based approach, by creating a web interface that developers with no experience in robot development can use to design and develop algorithms to remotely control different robotic platforms.

Contents

Figure 7.1: TODO

7.1 THE PMK USE CASE

In this section we give a detailed description of a test use case where a model-based approach has been used to replicate and reimplement an existing architecture developed with traditional techniques. We decided to start from an already implemented and fully functional system, to show that is possible to achieve the same level of functionalities of the original application with a model-based design combined with automatic programming.

The target robot is an electric wheelchair modified to be controlled with a computer, and equipped with various sensors to achieve levels of autonomy and teleoperation. The wheelchair used as the starting platform is a commercial model (*Twist T4 2x2*) produced by Degonda Rehab SA; it is suitable for both indoor and outdoor usage and it has high manoeuvrability thanks to the two-wheeled dynamics. The conversion from traditional electric wheelchair to autonomous robotic platform is achieved using the Personal Mobility Kit (PMK), which consists in four elements.

- Encoders connected to the electric motors controlling the wheels, they provide odometry information.

- Two *Sick TiM 561* laser scanner distance sensors, they provide a 360-degrees coverage around the wheelchair. They are used to map the environment, assist in the localisation of the robot and detect unexpected obstacles.
- *Shuttle DS81L*, an high-performance slim PC specifically designed for automotive and robotics applications. It is the on-board computer of the robot, it runs ROS and all the application code.
- The software components necessary to achieve the assisted and autonomous functionalities.

The PMK is designed to be an add-on that is possible to mount over any existing electric wheelchair to convert it into an autonomous or semi-autonomous platform. Thus, of the architecture we will present, the only platform specific part is the interface used to interface with on-board electronics, everything else is a modular design that can be adapted to multiple physical platforms.

A commercial electric wheelchair supports manual control through a joystick placed on one of the armrest. While this is suitable for most of users, there are cases of severe disabilities where the patients can only partially or cannot operate the joystick. The objective of PMK is to extend the functionalities of an electric wheelchair to provide assisted and autonomous control. In particular, the implemented software supports four different drive modes.

MANUAL WITH PMK OFF – This is the native configuration of the wheelchair, the user controls the movements directly with the on-board joystick. It is important to maintain the original operational mode even after the modification introduced by the PMK, the electric wheelchair should remain completely functional and operative even if an hardware or software malfunction causes the PMK to stop working. This act as a fallback emergency configuration.

MANUAL WITH PMK ON – In this configuration the PMK is active, but the user is still completely in control of the wheelchair. When mediated by the software system, the robotic platform can be controlled remotely by using a wireless joypad or directly with

the on-board joystick, a priority system ensures that the input from the joystick always has precedence over teleoperation. This is the neutral configuration of PMK, when the system is active, but not performing any action. This mode is particularly useful during the set up phases of the autonomous wheelchair (e. g., environment mapping).

ASSISTED – In this mode the PMK is enabled and actively meditates the commands coming from any input device. The aim of this configuration is to help the user operate the wheelchair by avoiding obstacles. Set-points sent from the wireless joypad or on-board joystick are processed and, if necessary, modified to avoid obstacle perceived by the two on-board laser scanners. As for the previous mode, the commands coming from the joystick have the priority over the wireless joypad.

AUTONOMOUS – Here the wheelchair is fully autonomous, the PMK is in charge of controlling the movements. The user can request a specific goal (e. g., "take me to the kitchen"), then the robotic platform will automatically reach the destination avoiding any obstacle along the way. For safety reasons, it is always possible to override the commands sent by the PMK in autonomous mode using the on-board joystick.

These are all the main functionalities of the electric wheelchair when equipped with the Personal Mobility Kit. Our aim is to create a system designed and developed using a model-based approach combined with automatic programming that exploits all the problem-related implementations already created for the original architecture and combine them with automatically generated ROS nodes, while maintaining the same functionalities.

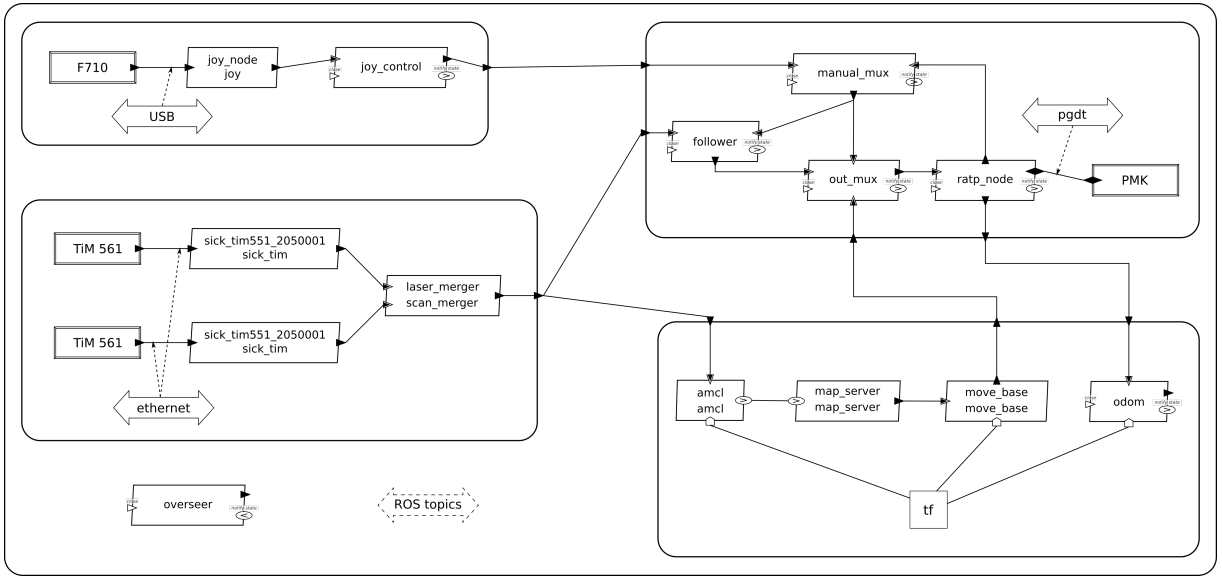


Figure 7.2: TODO

MODEL

The first step of the design of an architecture is to create the model. We did so by following the definitions and meta-models introduced in Chapter 4. Figure 7.2 shows a graphical representation of the complete model of the architecture of the robotic wheelchair. To organise and generalise the architecture we exploited the nesting capability of AADL by creating a hierarchical structure of system of systems. Each subsystem capture a specific functionality of the robot, and they are meant to be modular parts of the design that can be replaced when necessary. For example, going from laser scanner to cameras, or replacing the navigation system based on *move_base* with a custom one, or using a different electric wheelchair as platform.

TELEOPERATION – Similarly to the example presented in Section 4.3.2, this subsystem captures all the hardware and software components necessary to implement teleoperation. This specific models includes an AADL device modelling the wireless joypad used to control the wheelchair, a Logitech Gamepad F710. Directly connected to it, the device driver, the existing ROS node *joy_node* from the package *joy*; it reads the input from the joypad and converts it to ROS messages. The connection between the device and the driver is bound to an AADL bus modelling the physical USB interface between the joypad and the computer. The last software component in the subsystem is a node specifically designed for this application, *joy_control* is in charge of converting the messages coming from the device driver to velocity set-points to directly control the wheelchair (i. e., from *Joy* to *Twist* messages). Additionally, this node controls the global state machine of the system that selects the current driving mode (i. e., one of the four presented in Section 7.1), given the criticality of this communication it does not happen through ROS topics or services, but by accessing directly a shared memory area. This behaviour is captured in the model using a require data access port on the process modelling *joy_control*. On its frontier, the system exposes an outgoing port associated with the velocity command to relay the set-points generated by the teleoperation subsystem to the

rest of the architecture; moreover it has a data access to bridge the connection coming from the driving mode selection node (i. e., *joy_control*) to the shared memory area hosting the global state machine. Encapsulating the model in a system increases the modularity of the design, since it defines exactly the expected interfaces of the subsystem and makes it replaceable with a similar configuration (e. g., a different physical input) without altering the entire architecture.

SENSING – For the navigation and obstacle avoidance to work correctly, they need a single measurement from the laser rangefinder. Since the robotic platform is equipped with two sensors, the aim of the sensing subsystem is to unify their measurements in a single output; for this reason only one outgoing port is present on the system frontier representing the merged scan topic. Here there is a clear example on how multiple instances of same element can coexist as subcomponents. There are two occurrences of the same AADL device modelling the physical sensors mounted on the wheelchair, consequentially, the device driver component is duplicated too. As for the teleoperation subsystem, the connection between each laser scanner and the corresponding driver is bound to a model of a physical bus; but since the communication goes through an Ethernet connection it is a different component. All the connections converge in a single node in charge of merging the measurements coming from each laser rangefinder, that are then relayed outside the subsystem through the output port present on the frontier. In this system there is no custom node, since all of them come from existing packages or legacy implementations.

NAVIGATION – This subsystem mostly contains legacy nodes from ROS Navigation. A node implementing adaptive Monte Carlo localisation to estimate the position of the robot in a known environment (*amcl*), a node to store and share the current map of the environment (*map_server*), and the main navigation node (*move_base*) implementing global planning on the known map and local planning on the map generated in real time. To work, ROS Navigation requires laser measurements and odometry information, the former comes from the sensing subsystem, while the latter is generated by the custom *odom* node included in the navig-

ation subsystem; it receives the current speed from the wheelchair and uses it to estimate the position of the platform. The odometry node supports two types of output, as visible from the features of the AADL process, an *Odometry* message, published on a topic and modelled using an output port, and shared using *tf*, modelled thorough a data access directly connected to the data component representing the centralised collection of all the reference frames. Since it is the backbone of ROS Navigation, and it is used only by nodes in this subsystem, it was reasonable to model the data component representing *tf* here. Differently from the two previous subsystems (i. e., teleoperation and sensing), here there is no need to model any physical bus since this is a pure software system. The frontier of the navigation subsystem is more complex, since it has an input port for the laser measurements, another input port for the speed of the wheelchair and an output port for the set-point generated by *move_base*. Abstracting the structure and interface of the navigation subsystem is particularly important since it is the part of the architecture that is more subject to changes, it is common to use the same platform to test different algorithms and architectural solutions for navigation.

PLATFORM – All the hardware and software components related to the physical platform are contained in this subsystem, since they are all platform-related, each node is a custom design created specifically for this architecture. There are two hardware components: a device to model the interface between the architecture and the on-board electronics manufactured by Penny&Giles Drive Technologies Ltd. (PGDT), and the bus representing the special hardware connection used to exchange wheelchair-specific messages. This bus and the corresponding custom device driver (*ratp_node*) act as a bridge between platform-specific data streams and ROS messages. Given the multiple driving modes of the robot where two manual inputs, potentially mediated by the system and with different priorities, coexists, together with an autonomous mode, the architecture uses a combination of two multiplexers. One is used to select the specific manual input (i. e., *manual_mux*), while the other (i. e., *out_mux*) differentiate between the different driving modes (i. e., manual, assisted and autonomous). The

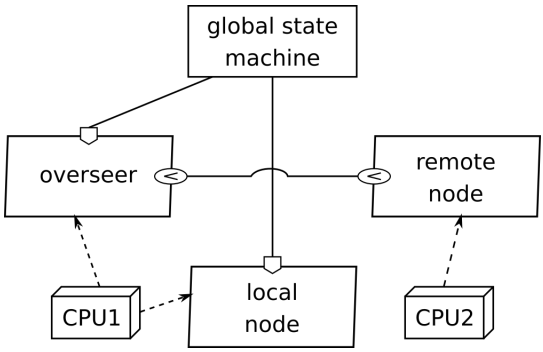


Figure 7.3: Graphical representation showing how two different nodes interact with the global state machine. TODO

two multiplexers have different behaviours, *manual_mux* is based on priorities, the input coming from the the on-board joystick will always override the wireless joypad, while *out_mux* changes the output depending on the current mode of the system, for this reason this component has a data access connected to the shared memory area hosting the global state machine. Finally, the last component is a node implementing local obstacle avoidance for assisted driving. Even with its peculiar configuration, the whole subsystem is designed with a general frontier, it exposes two inbound ports for the velocity commands (i. e., joypad and autonomous set-points) and one for scan messages (i. e., measurements for obstacle avoidance). The output port relays the current velocity of the wheelchair from the custom device driver node to the navigation subsystem. Additionally, as for the teleoperation subsystem, the platform system has a data access on its frontier to connect to the global state machine. By encapsulating the platform in a system, it is possible to replace it with a different wheelchair, or even with a completely different robotic platform.

MAIN SYSTEM – To increase clarity and use the model to evoke the modularity of the architecture, we modelled it with different subsystems. However, in AADL it is necessary to define a root system to encapsulate everything, moreover, some part of the architecture are too general to be fitted in a specific subsystem.

For this architecture, three components fall in this category: the *ROS communication* virtual bus, the *overseer* node and the global state machine. In the same way as physical buses are used in the teleoperation and platform subsystems, the *ROS communication* virtual bus is necessary to identify which connections model ROS communication protocols (i. e., topics or services), this is done by binding them to it. As originally introduced in Section 4.3.1 and, later, detailed in Section 5.2, the engineered nodes are designed with a strict life cycle, and they natively support a notification system to monitor the evolution of their state. The *overseer* node is in charge of collecting all the notification coming from the custom nodes, moreover it has a list of all the critical nodes (i. e., nodes that are subject to a liveness check) and trigger a transition on the global state machine if one of them stops working. The last element is the data component representing the shared memory area containing the global state machine. In the implementation, the global state machine is instantiated by the *overseer*, while all the nodes in the architecture can read or modify the current state by accessing the shared memory directly, it is also possible to interact with the global state machine through a ROS service interface mediated by the *overseer* (see Figure 7.3). The different approach used can be encoded in the model (i. e., connected through a data access or a subprogram call) however, in the implementation, it is determined at deployment time. In practice if the node runs on the same physical machine of the global state machine, then a shared memory approach is used, otherwise the interaction happens using ROS.

AUTOMATIC CODE GENERATION

In the previous section, we gave an overview of the model of the architecture by describing in details the subsystems and which functionalities they evoke. While the graphical representation of Figure 7.2 provides a general understanding of the topology of the architecture and gives some insights on the nature of the components, it does not capture all the details necessary to correctly perform automatic code generation. First of all, nodes belongs to three main categories: already implemented nodes, they are legacy

nodes previously developed, they may come from the standard ROS repositories (e. g., *move_base*) or be part of the original architecture (e. g., *laser_merger*), custom nodes, they are completely modelled and will be automatically generated (e. g., *odom*), special custom nodes, they are modelled correctly, but they contain unique implementation patterns that are not supported by the code generator (e. g., *ratp_node*). This last category will be covered in details in Section 7.1.3.

As discussed in Section 4.3.2, managing existing nodes in the architecture model is extremely important, since they are an invaluable resource provided by the ROS community. During the automatic code generation, existing nodes are ignored, since they do not provide any internal implementation but are described only as interfaces, however, they are correctly included in the generation of launch files describing the current configuration of the system. Moreover, connections between components can be specialised using properties to exploit topic remapping and define the runtime name of each topic, included those of already implemented nodes. Currently, the code generator does not support the use of neither ASN.1 nor JSON schema to define the parameters of the legacy nodes, however, it is possible to use properties to specify a standard ROS YAML file to be automatically included in the launch file as the parameter configuration of a specific node.

To automatically generate custom nodes that are compilation-ready, it is necessary to specify a series of properties. On a basic level, the designer can tune the behaviour of publisher and subscriber by setting the frequency, the queue size or the default name of the topic. Moving to a domain-specific point of view, the developer has to provide the implementations of the nodes; for this reason we decided to base our test use case on an existing architecture to exploit already implemented problem-specific code for the automatic programming phase. To understand how the process works in practice, let us take the example of the *follower* node. Without consider the domain-specific implementation, this component evokes a combination of a *sink* behaviour, the subscriber receiving the messages from the laser rangefinder, and a *filter* behaviour, manual set-points are received, modified according to the scan measurements, eventually modified and published.

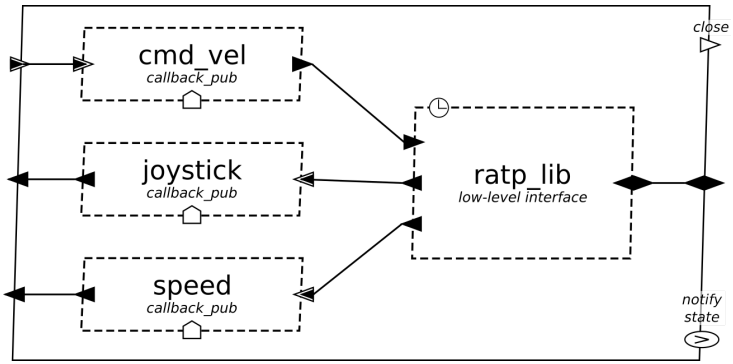


Figure 7.4: Simplified (i. e., focusing only on custom defined thread) graphical representation of the AADL description modelling the ratp_node

This configuration is translated to the code generator to two subscribers (i. e., one for the laser and one for the input commands) and a publisher (i. e., the output set-point). Moreover, by parsing the JSON files specified as properties in the component definition, the code generator can create the internal state of the node and set default and initial values for parameters and variables. After generating all the necessary source files, it is possible to include the domain-specific implementation as an external library, defined again in properties of threads and subprograms; this is done by carefully generating all the build files that bind together the auto-generated node skeleton and the manually implemented problem-specific code. When all the file related to the node are fully generated (i. e., core node implementation, internal state, external libraries, and build configurations), it is time to create the launch files. Since the *follower* node is part of the platform subsystem, it will be included in the platform-specific launch file, here the runtime configuration of the node is generated from the JSON files and included, moreover, as for the legacy nodes, the topics name are remapped according to the properties defined on the connections in the model. At this point, the generation of the node is complete and it is ready to be compiled and run.

SPECIAL NODES

Since all the problem-specific logic was already implemented in the original architecture, every element of the new system is generated using the automatic programming approach. There is only one exception: the *ratp_node*; this node is the device driver connecting the low-level electronics of the wheelchair to the ROS-based architecture. It implements a unique interface between ROS messages and custom data streams defined by the PGDT bus, therefore it is very challenging to automatically generate the source code using an automatic programming approach. Since this node is an hybrid between a ROS-based (i. e., receiving set-points and publishing the current speed and status) and an hardware-specific (i. e., interfacing with the PMK) implementation, we can partially exploit the code generator to define a node skeleton to be refined by the developer. There are two key challenges when using this approach: first, to have a model expressive enough to capture the peculiar design of the component, and second, to generate a node in such a way that the hardware-specific functionalities can be integrated without major modifications.

To tackle the first challenge we can exploit the expressive power provided by AADL. When not extending our base templates (see Section 4.4) used to identify ROS elements, AADL connections, ports and threads can be used to model any kind of communication protocol or execution flow. Figure 7.4 shows a simplified (recurring elements of the base ROS node are omitted) graphical representation of the model of the *ratp_node*. It is possible to use a periodic AADL thread together with a bidirectional port on its frontier to model the communication between the component and the low-level hardware. Additional ports model the exchange from a problem-specific implementation to the ROS communication protocols. The *ratp_lib* thread produces two outputs that are converted to ROS messages: one is the current set-point provided by the on-board joystick, and the other is the current speed of the wheelchair. Moreover, it receives one input from the rest of the system that is then converted to a format compatible with the low-level hardware: the set-point of the current active input. These port on the frontier of the low-level communication thread

are connected using internal connections to the corresponding threads modelling the publishers and subscribers. However, there is a significant distinction between this model and how ROS thread are usually modelled. Threads behaving as subscribers are not triggered by a message coming from outside the component, but directly by the *ratp_lib* thread, moreover, the publisher thread output does not relay its message to the rest of the architecture, but the output port is connected directly to the low-level communication thread.

The second challenge is implicitly solved by the design of the engineered node and how unknown subcomponents are managed by the code generator. As explained in Section 5.4.2, the code generator parses a process model and implements it as a ROS node, only if it extends the base ROS node model, moreover, of all the threads defined as subcomponents those that do not extend one of the predefined functionalities (i. e., publisher, subscriber, subscriber with publisher or service) are ignored and not processed by the automatic programming system. In the case of the *ratp_node* it means that the code generator recognises the component as a ROS node and generates all the basic structure, additionally, the three thread modelled as ROS elements (i. e., *cmd_vel*, *joystick* and *speed*) but connected to the *ratp_lib* thread are correctly generated as ROS subscribers or publishers. In summary, given the model presented in Figure 7.4, the resulting generated node extends the engineered node and correctly implements the life cycle, the asynchronous spinner, the encapsulated internal state and the separation between the middleware and the problem-specific code; moreover it contains three publishers and three subscribers. Out of these six ROS functionalities, three are already correctly implemented: the publisher for the current speed and the current command of the on-board joystick, and the subscriber for the external set-point. The other three needs to be converted from a ROS-based design to an implementation compatible with the low-level interface. While this approach may appear as an overcomplication, it actually streamlines the work of the developer. The code generator handle automatically all the ROS-related boilerplate, while defining a structure that can be easily exploited; the developer only needs to focus on implementing the thread

interfacing with the low-level hardware. In our implementation, we use the already created structure as a reference and replaced the callback structure of ROS with standard bindings. Basically, from the point of view of the node, the interaction still happens through a system of callbacks, however, they are not triggered by ROS, but controlled by the *ratp_thread*.

In conclusion, while this node requires a direct intervention of the developer to modify its structure, it is more efficient to model it, generate the code to obtain a partially implemented component and then manually insert some modifications, instead of implementing it from scratch. This is possible because of the modularity and flexibility of the engineered node, and because ROS is still the main element of the component design. While our code generator currently cannot completely and successfully process this type of nodes, this mixed approach (i. e., code generation combined with heavy human intervention) gives us an insight on how future nodes with the same characteristics could be automatically generated.

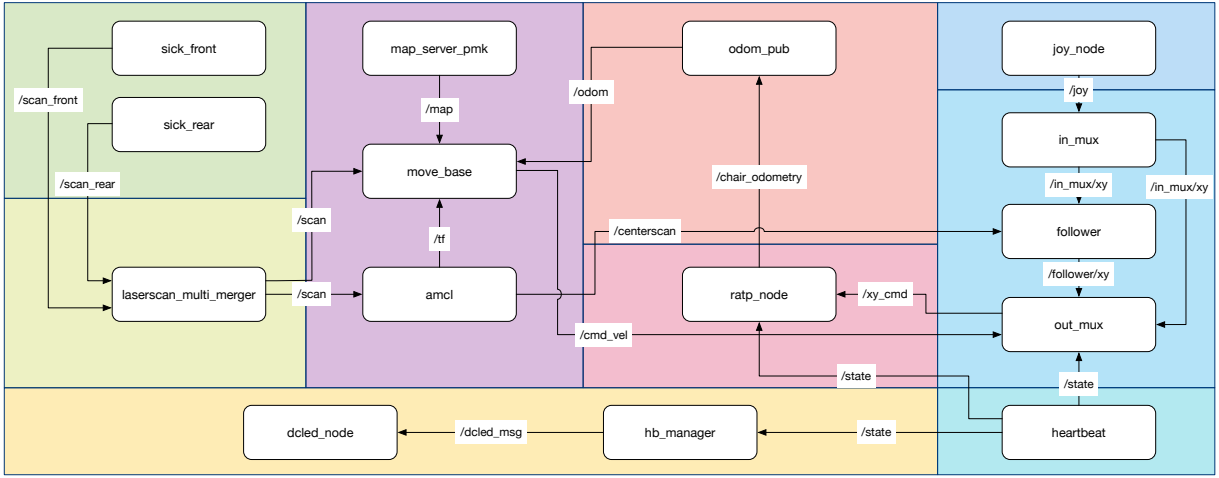


Figure 7.5: Original design of the architecture of the autonomous wheelchair.

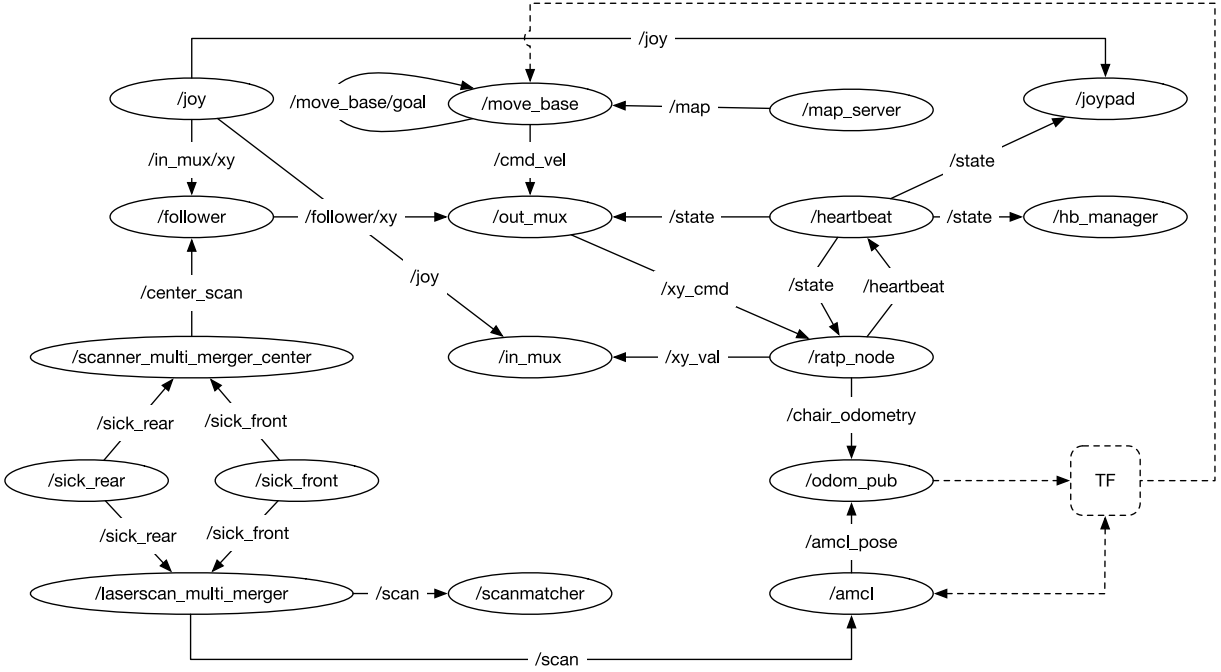


Figure 7.6: Runtime ROS graph of the hand-written architecture.

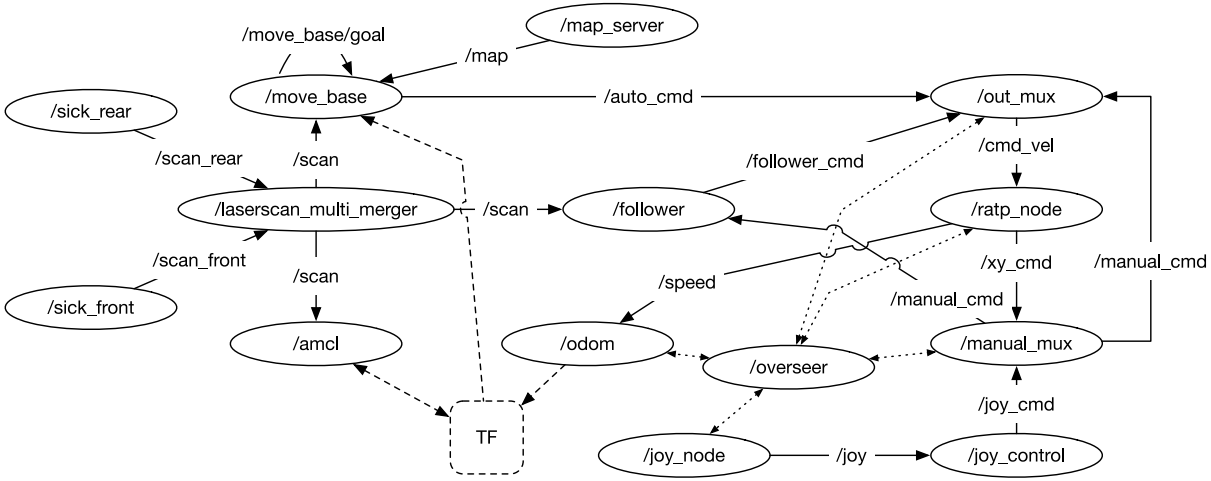


Figure 7.7: Runtime ROS graph of the automatically generated architecture.

COMPARISON

The aim of this use case was to showcase how a complete and functional architecture can be designed and developed entirely using a model-based approach combined with automatic code generation. Starting from an existing architecture was useful for multiple reasons: clear use case and requirements for the robot, existing implementations for all the problem-specific features, precise target functionalities, reference benchmark for the behaviour of the autonomous wheelchair. It was important for us to be able to compare the result of our approach to an existing implementation. In this section, we will analyse the original architecture to show the parallels with the automatic generated one and to highlight existing problems that were solved using a model-based design approach.

First of all, we have to compare the design of the architecture of the wheelchair according to the documentation (Figure 7.5) with the runtime computation graph generated by ROS (Figure 7.6). With a quick analysis it is possible to see that they do not match, some nodes mentioned in the documentation are not present in the graph (e. g., *dcled_node*) or vice versa (e. g., *scanner_multi_merger_center*), while others have been replaced (e. g., the standard *map_server* in place of the custom *map_server_pmk*). This is not surprising, the PMK is an ongoing research project where multiple people with varying experience contributed to it. As a result, features were added directly to the codebase without propagating them in the documentation or in the original design. As the project progressed, some of the features were removed, but they left some dangling dependencies behind, in the form of design choices and components. This problem is exacerbated by the fact that ROS does not provide any tool to visualise and analyse the complete architecture before runtime, the only way is to run the entire system and then view the ROS graph using tools like *rqt_graph*. However, even this approach is limited, since it only shows topics, but no services or actions. Of course, a model-based approach does not automatically solve all these problems, since a developer can, and sometimes needs to, directly modify the codebase; however, by combining models and automatic code

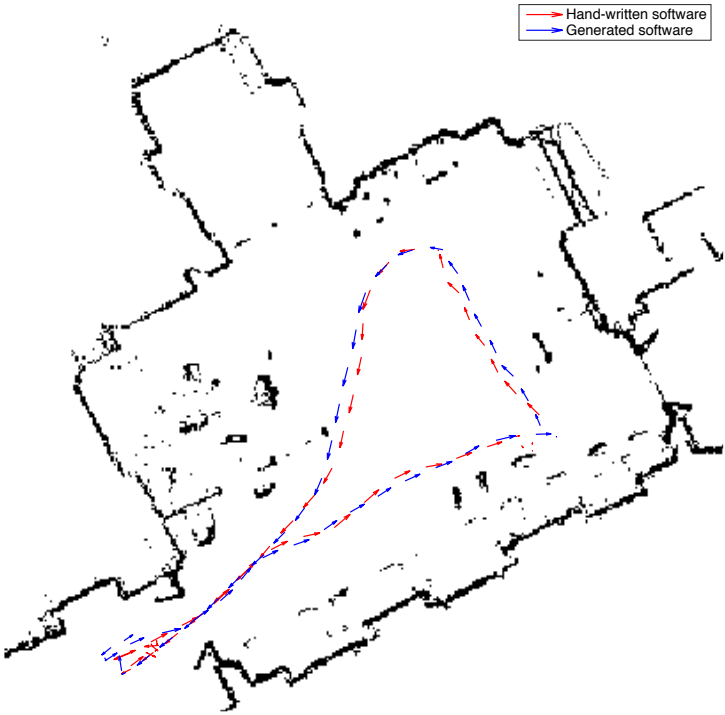


Figure 7.8: Comparison between the trajectory followed by the robot equipped with the hand-written software (in red) and the automatically generated implementation (in blue).

generation we can create an environment where good practices and designs are encouraged and more natural.

With a deeper analysis of the runtime graph it is possible to identify some architectural problems, that were not present in the original design but were introduced in sequential iterations of the project. There are three fundamental issues that could cause unexpected behaviours of the robot.

- There is a circular dependency between *odom_pub* and *amcl*. Both nodes are in charge of estimating the position of the robot, the former uses velocity measurement coming from the wheelchair, while the latter matches laser rangefinder

data with a know map. The circular dependency happens because each of them expect from the other an initial position to start the estimate. Given how the two algorithms are implemented, an incorrect starting position would not compromise the correct functioning of the system, however it may cause unexpected behaviour, especially after the initial start up of the system.

- The odometry is estimated twice in the system. This is not directly visible from the graph presented in Figure 7.6, since it is not detailed enough, however, the *rapt_node* pre-compute the odometry of the platform and provide it, together with the velocity measurements, to *odom_pub*. The initial computation is then discarded and recalculated again directly from the velocity. While this causes no malfunction, it is not a consistent design choice to do a specific processing twice in the same architecture.

- The management of the laser scanner measurements is flawed in multiple ways. As explained in Section 7.1.1, the navigation subsystem requires an unified source of laser information. In the original documentation, there was a single node in charge of merging the two laser sources, however, in the actual architecture, there are two nodes with the same task: *scanner_multi_merger_center* and *laserscan_multi_merger*. Moreover, it is not visible from the computation graph, but the lunch file of the original architecture included a third laser scan merger node, which it is killed at start up since it has the exact same name of another one, and ROS forbid it. Finally, there is a *scanmatcher* node that subscribe to a topic but has no role in the architecture, a leftover dependency from a removed functionality. Since the duplicated nodes are functionally identical, the overall functionality of the architecture are preserved. However, such a configuration is a waste of computational power and a potential source of significant and dangerous inconsistencies.

For comparison, Figure 7.7 shows the runtime graph of the architecture resulting from the model-based design and automatic

code generation. The overall structure of the system is very similar to the original architecture, however, all the issues identified before have been solved.

- There is no circular dependency between *amcl* and *odom*. They receive the initial position independently as a parameter or through an external topic.
- The odometry is estimated only once in the *odom* node. The *ratp_node* provides directly and only the current speed of the wheelchair as a *Twist* message.
- Only one node is in charge of unifying the laser sources and the *scanmatcher* node has been removed.

Analysing the ROS graph it is useful to provide an overview of the quality of the design of the architecture, however, it gives no insight on the actual functionalities of the system. As seen from the original architecture, design flaws not always translate in a compromised system or faulty behaviours. The robotic wheelchair equipped with the handcrafted architecture supported full autonomous driving with no serious issues, except for minor problems caused by an unstable communication with the low-level control module. This means that we can use the behaviour of the original architecture to provide an empirical proof of the correctness of the generated architecture. We performed a comparison between the path followed by the wheelchair when running the original system in autonomous mode, and then we gave the same goal to the generated architecture; Figure 7.8 shows the result. The resulting paths are extremely similar, this shows that not only the generated architecture replicates the same results of the original one, moreover, all the problem-specific implementations have maintained their original functionalities even after they are transposed in the automatically generated ROS environment.

7.2 WEB INTERFACE

By decoupling the *Server* from the *OntoRob Interface* we can create multiple interfaces that can access the functionalities of the robot

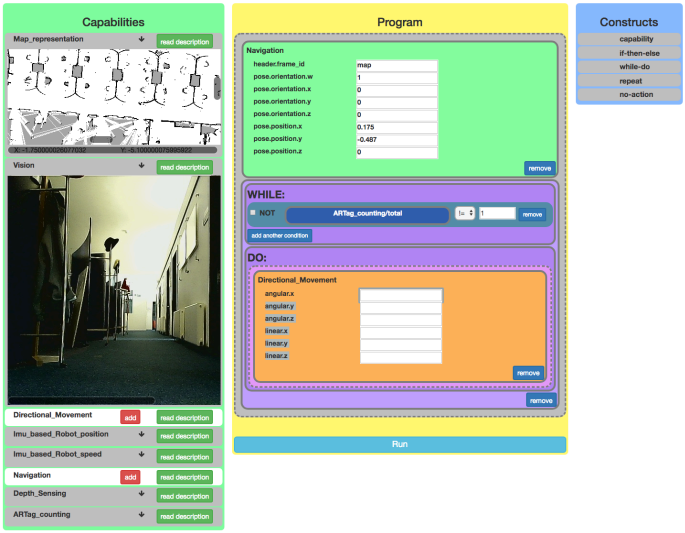


Figure 7.9: The web interface used to interact with the capabilities evoked by the robot.

through the capability system. In an effort to explore different approaches, push the boundaries of our implementation, and test it directly with users in a controlled environment, we developed an additional *Web Interface*. This interface, an instance is visible in Figure 7.9, replaces the *OntoRob interface* and interact directly with the *Server*.

Once set up, this interface is accessible from any web browser, it was specifically designed to be simple, intuitive, easy to understand, and accessible. The aim is not to implement a complete interface to program and control remotely a robot, but to create a proof-of-concept remote interface that can be used to inspect the status of the robot (through *read*-capabilities) and give a sequence of simple command (through *write*-capabilities). To make the interface more complete, we implemented a simple programming language to chain multiple commands together and check condition based on the output of *read*-capabilities.

In this section, we give a description of the graphical user interface presented by the *Web interface*, then we show the results

obtained in an experiment where a set of user with no previous experience in robotics or ROS had to develop programs using the interface to complete simple robotic task with two different platforms.

GUI DESCRIPTION

The *Web interface* is designed to be essential and to show as much information as possible in without overloading the user. It is divided in three different panels, each one focusing on a specific aspect of our approach: the capability panel, it presents the functionality of the robot in a compact way, the program panel, it is the main part of the interface and it contains the program created by the user, and construct panel, it is the "toolbox" the user has to create the program.

CAPABILITY PANEL – This panel presents to the user the capabilities abstracted by the combined work of the *Analyzer* and the *Server*. When activated, the *Web interface* contacts the *Server* and retrieve the list of all the currently active and accessible capabilities. The retrieved list is presented to the user in various forms depending on the nature of each capability. First of all, the distinction between *read*-capabilities and *write*-capabilities. The former represent the flow of information coming from the robot to the user, therefore their value is constantly updated and visualised in the GUI; for example, the box connected to the *Robot position* capabilities contains three fields (i. e., x, y and z coordinates) showing the current position of the robot in space, the *Web Interface* periodically queries the *server* to update the content of the fields with the most recent position. The latter represent the entry points a user can exploit to interact with the robot, therefore they are presented only as a list of available capabilities. For each capability the most suitable visualisation system is used, for instance, the velocity of the robot is more understandable in numerical form showing the raw content of the original ROS message, while the video feed of the on-board camera is better is rendered to show the actual pictures. Each capability is associated with a description to help

the user understand and interpret the information provided, or how to exploit the entry point to interact with the robot.

PROGRAM PANEL – The central area of the *Web interface*, as seen in Figure 7.9, is dedicated to the user to create program to send to the robot. The user can select each sub-panel of the programming area to interact with it. By selecting the outermost panel (i. e., the grey border around the whole program in the figure) the user can add a new capability or a new programming paradigm, while by selecting internal panels (e. g., the pink border in the DO panel) the user can define subroutines and specify conditions. When added to the *Program panel*, a capability will show the available parameters as fillable fields, so the user can specify the actual value to send to the robot. Once the user is satisfied with the set of commands, he can send them to the robot using the *Run* button. When the button is pressed, the commands are converted by the *Web interface* in a JSON file and sent to the *Server*. The *Server* decode the list of instructions and execute them in the necessary order through the *Dynamic node*.

CONSTRUCT PANEL – The rightmost panel of the *Web interface* represent the "toolbox" of the user. We did not want to implement a complex and fully functional programming language, therefore we focused on the most essential construct to give the user enough freedom. The first on the list is the capability, by using this construct the user can add a special container to the program, then by selecting the container can add a specific *write*-capability by using the *add* button from the *capability panel*. Then there are all the constructs available to change the control flow of the program: *if-then-else*, for a conditional jump, *while-do*, for a condition-based loop, and *repeat*, to repeat a set of instructions a specific amount of times. A program can be created as an imperative programming language, in which the atomic blocks are invocations of the available robot capabilities, and any conditional operator. An additional *no-action* construct can be used to perform a no operations, for example for an *if-then-else* with an empty *else* statement, or to create waiting loops. The parameters of a capability can be used in the conditions, so to exploit any robot output to drive the program flow (e. g., moving forward until an object is detected).

MODE	s-VARIANT	r-VARIANT
write	autonomous navigation directional movement	take-off land directional movement
read	vision current position current speed map representation object recognition	vision current position object recognition

Table 7.1: Robot capabilities for the two exercise variants.

EXPERIMENTAL SETUP

Four exercises of increasing difficulty were asked to be performed by each user, which corresponded to creating a program allowing a robot to achieve a specific task. In order to demonstrate that the ontology-based system could allow the abstraction of robot capabilities independently from the platform, we set up two variants of each exercise, a simulated one with a ground wheeled robot operating in an office environment (s-variant), and a real-world one with a drone flying in an indoor space (r-variant). Table 7.1 presents the robot capabilities available in each setting.

EXERCISE 1: SINGLE COMMAND – The first exercise requires the user to send a single command to the robot (i. e., exploit a single *write*-capability). The exercise was designed to let the user familiarise with *Web interface* by focusing only on understanding the concept of capability and how to use them to operate the robot. In the s-variant, the user needs to exploit two capabilities: *Map representation* to understand the current position of the robot and select a potential destination, and *Autonomous navigation* to command the robot to reach the specified destination. For the r-variant, the exercises starts when the drone already performed a successful take-off (i. e., the drone is already flying), therefore the user needs

to use only the *Directional movement* capability to move the drone in any direction.

EXERCISE 2: COMMAND SEQUENCE – The second exercise requires the user to chain together a list of commands to send to the robot (i. e., exploit multiple *write*-capabilities or the same one multiple times). The objective of this exercise is to let the user know that he can create a complex behaviour for the robot, and not only send atomic actions. The `s`-variant is a direct extension of the previous exercise, since the user needs to instruct the robot to navigate to two different locations, one after the other. In the `r`-variant, the exercise will start again with a drone that has successfully performed a take-off, the user has to instruct the drone with any motion command, then command the drone to land using the *Land* capability.

EXERCISE 3: CONDITION-BASED HALT – The third exercise requires user to implement a sequence of actions with a termination condition (i. e., use a condition to manage the execution of a *write*-capability). In this exercise we introduce to the user the first conditional constructs (e. g., *repeat* and *while-do*), and we let him discover that he can create complex programs. In the `s`-variant, the robot need to patrol three different locations, stopping only once all the locations had been visited at least twice. This requires to exploit the *Autonomous navigation* capability and to user the *repeat* construct. In the `r`-variant, the user has to instruct the drone to keep on turning on itself until it has performed at least a ration of 180° on itself, at that point the drone has to perform a landing. In this case the user has to exploit both *write*-capabilities to pilot the drone (i. e., *Directional movement* and *Land*) and a *read*-capability (i. e., *Robot orientation*) in conjunction with a conditional statement to complete the exercise.

EXERCISE 4: OBJECT RECOGNITION – In the final exercise, the user has to combine base capabilities (e. g., *Directional movement* or *Navigation*) with advanced capabilities based on additional functionalities introduced in the robot. In particular, the user had to create a behaviour for the robot based on the detection of ARtags. The aim of this exercise is to show to the user that

is possible to extend the functionalities of the robot with extra capabilities and to increase the complexity of the program. In the s-variant, the robot has to patrol a set of locations until an ARtag is detected. In the r-variant, the user had to implement three different movement behaviours for the drone each of them triggered by a different ARTag.

RESULTS AND DISCUSSION

A total of fourteen users were involved in the evaluation, equally shared between the s- and the r-variant. All the users had at least some basic programming knowledge, however, none of them has any robotic background or any previous experience with ROS or any other robotic middleware or framework. As a starting point, users were first allowed to familiarise themselves with the interface, namely through clicking on the different sections to understand the general behaviour of the tool. To make the task of resolving the different exercises more challenging for the users, they were prevented from reading the description of the capabilities in this initial familiarisation phase. After this first step, they were asked to solve all four exercises one after the other. For every exercise, we measured the time from the end of the task description until the final execution of the program.

Table 7.2 shows the average time ($\bar{t}$) required by the users to solve each exercise, along with the average number of programming blocks ($\overline{pb}$) and the number of capabilities (*no. cap*) required to solve the task. All the exercises were successfully carried out by all users. As one can see, Ex. 1 took slightly longer (especially in the s-variant), when compared with other more complex exercises. This can be attributed to the time users required to familiarise themselves with the capabilities of the robot they were working with, which they did not know beforehand. The relatively high variance in the time taken for Ex. 4 is due to this particular exercise having multiple solutions, some of which taking longer to implement than others.

A key, straightforward conclusion from this table is that users of this tool, who had no experience of programming robots and no prior knowledge of the architecture of the robot they were

	Ex. 1	Ex. 2	Ex. 3	Ex. 4
		s-variant		
$\overline{pb}$	1	2	4	9.5
no. cap	1	1	1	2
$\bar{t}$	1:22 ± 42s	1:04 ± 23s	1:15 ± 16s	6:52 ± 1:46
		r-variant		
$\overline{pb}$	1	2	4	8
no. cap	1	2	4	4
$\bar{t}$	1:16 ± 3s	01:16 ± 8s	4:05 ± 15s	5:47 ± 1:39

Table 7.2: Results obtained by the non-experts for the s-variant and the r-variant.

manipulating, managed to successfully program such a robot to achieve tasks from the very simple Ex. 1 to the more difficult Ex. 4 in a matter of a few minutes. Considering the inherent complexity of robot programming and of understanding not only what a robot can do (what capabilities it possesses), but also how to use it (how to invoke those capabilities), this can be considered a non-trivial achievement.

A direct comparison with how the same users would have achieved the same tasks without the tool provided is not feasible and would turn out to be meaningless. However, it appears a straightforward assumption that, those users not being familiar with ROS, the simple (in our tool) process to understand the different components of the robot, what they do and, crucially, how to invoke them, would require more than a few minutes by itself. ROS is a complex framework, requiring hours of practice to master. In addition, analysing the runtime graph of the specific robot to understand which topics and services are being be used (i. e., what the tool does through the ontology) is far from an easy task. A number of ROS nodes would need to be implemented from

	Ex. 1	Ex. 2	Ex. 3	Ex. 4
		s-variant		
LOC	35	58	64	82
no. com	1	2	2	3
no. msg	1	2	2	3
		r-variant		
LOC	34	39	56	59
no. com	1	2	4	4
no. msg	1	2	3	3

Table 7.3: Results obtained by the expert for the s-variant and the r-variant.

scratch to encapsulate the required functionalities, and managing the correct publishers and subscribers. Lastly, the nodes would need to be deployed and integrated with the robot architecture. Knowledge of specific packages (e. g., *move_base* for autonomous navigation) is also required by some of the exercises. In other words, while a direct comparison could not have been performed, there is little doubt that significantly more effort would have been required to enable our non-expert users to achieve the same results with ROS, as it did with our tool.

As an additional point towards the validity of our claim that our tool reduces the effort required to exploit robots' capabilities and therefore make them more accessible, we asked an expert in robotics with extensive experience in ROS to achieve the same task. Once again, the objective here is not to compare the experts to the non-experts using two different frameworks, but to provide an intuitive understanding of the difficulty of realizing the tasks achieved by our users without our tool. In Table 7.3, we therefore show for each task in each variant:

- the number of lines of code used by the ROS expert (*LOC*),

- the number of ROS communication components (publishers and subscribers) employed (*no. com*),
- the number of message types used (*no. msg*).

These metrics give an estimate of the effort required by a ROS developer to solve the specified tasks. Lines of code set a lower bound for the implementation time, while number of components and messages outline the complexity of the solution.

This comparison further show how programming a robot is made "easier" and, through abstracting capabilities from the technical aspects of their implementation, requires less complexity. Our approach and the associated tool therefore represent a viable solution to enable non-expert users to exploit robots in ways that were before only accessible to expert ROS programmers.

8 CONCLUSIONS

I tried so hard, and got so far, but in the end it doesn't even matter.

BIBLIOGRAPHY

[1] AUTOSAR website. https://www.autosar.org/. [Online, accessed 10-October-2019]. 2019.

[2] Alain Abran, James W Moore, Pierre Bourque, Robert Dupuis and L Tripp. 'Software engineering body of knowledge'. In: IEEE Computer Society, Angela Burgess (2004).

[3] Roberto Acerbis, Aldo Bongio, Marco Brambilla, Massimo Tisi, Stefano Ceri and Emanuele Tosetti. 'Developing eBusiness solutions with a model driven approach: the case of acer EMEA'. In: International Conference on Web Engineering. Springer. 2007, pp. 539–544.

[4] Omar Aldawud, Tzilla Elrad and Atef Bader. 'A UML profile for aspect oriented modeling'. In: OOPSLA 2001 workshop on aspect Oriented Programming. 2001.

[5] Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard and François Terrier. 'Enabling scheduling analysis for AUTOSAR systems'. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE. 2011, pp. 152–159.

[6] Thomas Arts, John Hughes, Ulf Norell and Hans Svensson. 'Testing AUTOSAR software with QuickCheck'. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE. 2015, pp. 1–4.

[7] Gianluca Bardaro and Matteo Matteucci. 'Using AADL to model and develop ROS-based robotic application'. In: 2017 First IEEE International Conference on Robotic Computing (IRC). IEEE. 2017, pp. 204–207.

[8] Gianluca Bardaro, Andrea Semprebon, Agnese Chiatti and Matteo Matteucci. 'From Models to Software Through Automatic Transformations: An AADL to ROS End-to-End Toolchain'. In: 2019 Third IEEE International Conference on Robotic Computing (IRC). IEEE. 2019, pp. 580–585.

[9] Gianluca Bardaro, Andrea Semprebon and Matteo Matteucci. 'AADL for robotics: a general approach for system architecture modeling and code generation'. In:

[10] Gianluca Bardaro, Andrea Semprebon and Matteo Matteucci. 'A use case in model-based robot development using AADL and ROS'. In: Proceedings of the 1st International Workshop on Robotics Software Engineering. ACM. 2018, pp. 9–16.

[11] Johannes Baumgartl, Thomas Buchmann, Dominik Henrich and Bernhard Westfechtel. 'Towards Easy Robot Programming-Using DSLs, Code Generators and Software Product Lines.' In: ICSOFT. 2013, pp. 548–554.

[12] David Benavides, Pablo Trinidad and Antonio Ruiz-Cortés. 'Automated reasoning on feature models'. In: International Conference on Advanced Information Systems Engineering. Springer. 2005, pp. 491–503.

[13] Lorenzo Bettini. Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd, 2016.

[14] Geoffrey Biggs, Kiyoshi Fujiwara and Keiju Anada. 'Modelling and analysis of a redundant mobile robot architecture using aadl'. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots. Springer. 2014, pp. 146–157.

[15] Stéphane Bonnet, Jean-Luc Voirin, Daniel Exertier and Véronique Normand. 'Not (strictly) relying on sysml for MBSE: language, tooling and development perspectives: The arcadia/capella rationale'. In: 2016 Annual IEEE Systems Conference (SysCon). IEEE. 2016, pp. 1–6.

[16] Marco Brambilla, Jordi Cabot and Manuel Wimmer. 'Model-driven software engineering in practice'. In: Synthesis Lectures on Software Engineering 1.1 (2012), pp. 1–182.

[17] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams and Anders Orebäck. 'Orca: A component model and repository'. In: Software engineering for experimental robotics. Springer, 2007, pp. 231–251.

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

[19] Davide Brugali and Patrizia Scandurra. 'Component-based robotic engineering (part i)[tutorial]'. In: IEEE Robotics & Automation Magazine 16.4 (2009), pp. 84–96.

[20] Davide Brugali and Azamat Shakhimardanov. 'Component-based robotic engineering (part ii)'. In: IEEE Robotics & Automation Magazine 17.1 (2010), pp. 100–112.

[21] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault and Frédéric Madiot. 'MoDisco: a generic and extensible framework for model driven reverse engineering'. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM. 2010, pp. 173–174.

[22] Herman Bruyninckx. 'Open robot control software: the OROCOS project'. In: Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164). Vol. 3. IEEE. 2001, pp. 2523–2528.

[23] Herman Bruyninckx. 'OROCOS: design and implementation of a robot control software framework'. In: Proceedings of IEEE International Conference on Robotics and Automation. Citeseer. 2002.

[24] Herman Bruyninckx, Peter Soetens and Bob Koninckx. 'The real-time motion control core of the Orocos project'. In: 2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422). Vol. 2. IEEE. 2003, pp. 2766–2771.

[25] Stefano Ceri, Piero Fraternali and Aldo Bongio. 'Web Modeling Language (WebML): a modeling language for designing Web sites'. In: Computer Networks 33.1-6 (2000), pp. 137–157.

[26] Nitishal Chungoora, Robert I Young, George Gunendran, Claire Palmer, Zahid Usman, Najam A Anjum, Anne-FrançOise Cutting-Decelle, Jennifer A Harding and Keith Case. 'A model-driven ontology approach for manufacturing system interoperability and knowledge sharing'. In: Computers in Industry 64.4 (2013), pp. 392–401.

[27] IEEE Standards Coordinating Committee et al. 'IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos'. In: CA: IEEE Computer Society 169 (1990).

[28] Steve Cousins. 'Exponential growth of ros [ros topics]'. In: IEEE Robotics & Automation Magazine 1.18 (2011), pp. 19–20.

[29] James B Dabney and Thomas L Harman. Mastering simulink. Pearson, 2004.

[30] Enrico Daga, Mathieu d'Aquin, Alessandro Adamou and Enrico Motta. 'Addressing exploitability of smart city data'. In: Smart Cities Conference (ISC2), 2016 IEEE International. IEEE. 2016, pp. 1–6.

[31] Didier Delanote, Stefan Van Baelen, Wouter Joosen and Yolande Berbers. 'Using AADL to model a protocol stack'. In: 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008). IEEE. 2008, pp. 277–281.

[32] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi and Mikal Ziane. 'Robotml, a domain-specific language to design, simulate and deploy robotic applications'. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots. Springer. 2012, pp. 149–160.

[33] Alonso Diego, Cristina Vicente Chicote, Ortiz Francisco, Pastor Juan and Álvarez Bárbara. 'V3cmm: A 3-view component meta-model for model-driven robotic software development'. In: (2010).

[34] Dale Dougherty and Arnold Robbins. sed & awk: UNIX Power Tools. " O'Reilly Media, Inc.", 1997.

[35] Brian Elvesæter, Cyril Carrez, Parastoo Mohagheghi, Arne-Jørgen Berre, Svein G Johnsen and Arnor Solberg. 'Model-driven service engineering with SoaML'. In: Service Engineering. Springer, 2011, pp. 25–54.

[36] Magnus Eriksson, Jürgen Börstler and Kjell Borg. 'Managing requirements specifications for product lines–An approach and industry case study'. In: Journal of Systems and Software 82.3 (2009), pp. 435–447.

[37] Huascar Espinoza, Daniela Cancila, Bran Selic and Sébastien Gérard. 'Challenges in combining SysML and MARTE for model-based design of embedded systems'. In: European Conference on Model Driven Architecture-Foundations and Applications. Springer. 2009, pp. 98–113.

[38] Moritz Eysholdt and Heiko Behrens. 'Xtext: implement your language faster than the quick and dirty way'. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM. 2010, pp. 307–309.

[39] Madeleine Faugere, Thimothee Bourbeau, Robert De Simone and Sebastien Gerard. 'Marte: Also an uml profile for modeling aadl applications'. In: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007). IEEE. 2007, pp. 359–364.

[40] Peter H Feiler, David P Gluch and John J Hudak. The architecture analysis & design language (AADL): An introduction. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[41] Peter Feiler. 'Open source aadl tool environment (osate)'. In: AADL Workshop, paris. 2004, pp. 1–40.

[42] Marcus Fontoura, Wolfgang Pree and Bernhard Rumpe. The UML profile for framework architectures. Addison-Wesley Longman Publishing Co., Inc., 2000.

[43] Martin Fowler. Domain-specific languages. Pearson Education, 2010.

[44] Sanford Friedenthal, Alan Moore and Rick Steiner. A practical guide to SysML: the systems modeling language. Morgan Kaufmann, 2014.

[45] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. 'An introduction to UML profiles'. In: UML and Model Engineering 2 (2004), pp. 6–13.

[46] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa and Klaus Lange. 'AUTOSAR–A Worldwide Standard is on the Road'. In: 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden. Vol. 62. 2009, p. 5.

[47] Brian Gerkey, Richard T Vaughan and Andrew Howard. 'The player/stage project: Tools for multi-robot and distributed sensor systems'. In: Proceedings of the 11th international conference on advanced robotics. Vol. 1. 2003, pp. 317–323.

[48] Philippe Gerum. 'Xenomai-Implementing a RTOS emulation framework on GNU/Linux'. In: White Paper, Xenomai (2004), p. 81.

[49] Luca Gherardi and Davide Brugali. 'Modeling and reusing robotic software architectures: the hyperflex toolchain'. In: 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE. 2014, pp. 6414–6420.

[50] Holger Giese, Stephan Hildebrandt and Stefan Neumann. 'Model synchronization at work: keeping SysML and AUTOSAR models consistent'. In: Graph transformations and model-driven engineering. Springer, 2010, pp. 555–579.

[51] Ian S Graham et al. The HTML sourcebook. Wiley New York, 1995.

[52] Richard C Gronback. Eclipse modeling project: a domain-specific language (DSL) toolkit. Pearson Education, 2009.

[53] Giancarlo Guizzardi. 'On ontology, ontologies, conceptualizations, modeling languages, and (meta) models'. In: Frontiers in artificial intelligence and applications 155 (2007), p. 18.

[54] Elliotte Rusty Harold. XML: extensible markup language. IDG Books Worldwide, Inc., 1998.

[55] Jerome Hugues, Bechir Zalila, Laurent Pautet and Fabrice Kordon. 'From the prototype to the final embedded system using the Ocarina AADL tool suite'. In: ACM Transactions on Embedded Computing Systems (TECS) 7.4 (2008), p. 42.

[56] Internet Communication Engine. https://github.com/zeroc-ice/ice. [Online, accessed 11-October-2019]. 2019.

[57] Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink and Cetin Demir. 'Pragmatic and opportunistic reuse in innovative start-up companies'. In: IEEE software 25.6 (2008), pp. 42–49.

[58] Jetbrains MPS. https://www.jetbrains.com/mps/. [Online, accessed 10-October-2019]. 2019.

[59] Sylvain Joyeux and Jan Albiez. 'Robot development: from components to systems'. In: 6th National Conference on Control Architectures of Robots. 2011, 15–p.

[60] Holger Krahn, Bernhard Rumpe and Steven Völkel. 'MontiCore: a framework for compositional development of domain specific languages'. In: International journal on software tools for technology transfer 12.5 (2010), pp. 353–372.

[61] Pranav Kumar, William Emfinger, Gabor Karsai, Dexter Watkins, Benjamin Gasser and Amrutur Anilkumar. 'ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS'. In: Electronics 5.3 (2016), p. 53.

[62] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois and François Terrier. 'Papyrus UML: an open source toolset for MDA'. In: Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009). 2009, pp. 1–4.

[63] Morten Larsen. 'Modelling field robot software using aadl'. In: Technical Report Electronics and Computer Engineering 4.25 (2016).

[64] Sergio Luján-Mora, Juan Trujillo and Il-Yeol Song. 'A UML profile for multidimensional modeling in data warehouses'. In: Data & Knowledge Engineering 59.3 (2006), pp. 725–769.

[65] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione and Antony Tang. 'What industry needs from architectural languages: A survey'. In: IEEE Transactions on Software Engineering 39.6 (2012), pp. 869–891.

[66] Stephen J Mellor, Marc Balcer and Ivar Foreword By-Jacoboson. Executable UML: A foundation for model-driven architectures. Addison-Wesley Longman Publishing Co., Inc., 2002.

[67] Marcilio Mendonca, Moises Branco and Donald Cowan. 'SPLOT: software product lines online tools'. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM. 2009, pp. 761–762.

[68] Elisa Yumi Nakagawa, Pablo Oliveira Antonino and Martin Becker. 'Reference architecture and product line architecture: A subtle but critical difference'. In: European Conference on Software Architecture. Springer. 2011, pp. 207–211.

[69] Zainalabedin Navabi. VHDL: Analysis and modeling of digital systems. McGraw-Hill, Inc., 1997.

[70] Linda M Northrop. 'SEI's software product line tenets'. In: IEEE software 19.4 (2002), pp. 32–40.

[71] Object Management Group. https://www.omg.org/. [Online, accessed 09-October-2019]. 2019.

[72] Randy Otte, Paul Patrick and Mark Roy. Understanding CORBA: common object request broker architecture. Vol. 19. Prentice Hall PTR, 1996.

[73] A Perrotin, Eric Conquet, Pierre Dissaux, Thanassis Tsiodras and Jerome Hugues. 'The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software'. In: 2010.

[74] Maxime Perrotin, Eric Conquet, Julien Delange, AndrÚ Schiele and Thanassis Tsiodras. 'TASTE: a real-time software engineering tool-chain overview, status, and future'. In: International SDL Forum. Springer. 2011, pp. 26–37.

[75] Roger S Pressman. Software engineering: a practitioner's approach. Palgrave Macmillan, 2005.

[76] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler and Andrew Y Ng. 'ROS: an open-source Robot Operating System'. In: ICRA workshop on open source software. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[77] Spencer Rugaber and Kurt Stirewalt. 'Model-driven reverse engineering'. In: IEEE software 21.4 (2004), pp. 45–53.

[78] James Rumbaugh, Ivar Jacobson and Grady Booch. Unified modeling language reference manual, the. Pearson Higher Education, 2004.

[79] Hermann Schichl. 'Models and the history of modeling'. In: Modeling languages in mathematical optimization. Springer, 2004, pp. 25–36.

[80] Christian Schlegel and Dennis Stampfer. 'The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development'. In: Tutorial on Managing Software Variability in conjunction with the Robot Control Systems at Robotics: Science and Systems Conference (RSS). Berkeley, CA, USA, July. 2014.

[81] David Sciamma, Gilles Cannenterre and Jacques Lescot. Ecore Tools. Tech. rep. Technical report, last update: May, 2013.

[82] Bran Selic. 'The pragmatics of model-driven development'. In: IEEE software 20.5 (2003), pp. 19–25.

[83] Shane Sendall and Wojtek Kozaczynski. 'Model transformation: The heart and soul of model-driven software development'. In: IEEE software 20.5 (2003), pp. 42–45.

[84] Anthony JH Simons and Ian Graham. '30 Things that go wrong in object modelling with UML 1.3'. In: Behavioral Specifications of Businesses and Systems. Springer, 1999, pp. 237–257.

[85] Ian Sommerville. 'Software engineering 9th Edition'. In: ISBN-10137035152 (2011).

[86] Dave Steinberg, Frank Budinsky, Ed Merks and Marcelo Paternostro. EMF: eclipse modeling framework. Pearson Education, 2008.

[87] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze and Andreas Wortmann. 'A new skill based robot programming language using UML/P Statecharts'. In: 2013 IEEE International Conference on Robotics and Automation. IEEE. 2013, pp. 461–466.

[88] Ilaria Tiddi, Emanuele Bastianelli, Gianluca Bardaro and Enrico Motta. 'A user-friendly interface to control ROS robotic platforms'. In: 2018 ISWC Posters and Demonstrations, Industry and Blue Sky Ideas Tracks, ISWC-P and D-Industry-BlueSky 2018. CEUR. 2018.

[89] Ilaria Tiddi, Emanuele Bastianelli, Gianluca Bardaro, Mathieu d'Aquin and Enrico Motta. 'An ontology-based approach to improve the accessibility of ROS-based robotic systems'. In: Proceedings of the Knowledge Capture Conference. ACM. 2017, p. 13.

[90] Ilaria Tiddi, Enrico Daga, Emanuele Bastianelli and Mathieu d'Aquin. 'Update of time-invalid information in knowledge bases through mobile agents'. In: Integrating Multiple Knowledge Representation and Reasoning Techniques in Robotics (2016).

[91] Richard T Vaughan and Brian P Gerkey. 'Reusable robot software and the player/stage project'. In: Software Engineering for Experimental Robotics. Springer, 2007, pp. 267–289.

[92] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet and Fabrice Kordon. 'PolyORB: a schizophrenic middleware to build versatile reliable distributed applications'. In: International Conference on Reliable Software Technologies. Springer. 2004, pp. 106–119.

[93] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser and Guido Wachsmuth. DSL engineering: Designing, implementing and using domain-specific languages. dslbook. org, 2013.

[94] Niklaus Wirth. 'Algorithms and data structures'. In: (1986).

[95] Tewfik Ziadi, Loïc Hélouët and Jean-Marc Jézéquel. 'Towards a UML profile for software product lines'. In: International Workshop on Software Product-Family Engineering. Springer. 2003, pp. 129–139.

[96] lxml - XML and HTML with Python. http://lxml.de. [Online, accessed 15-November-2017]. 2017.