GIANLUCA BARDARO

ROBOTS, SOFTWARE AND OTHER STUFF

bip, bop.

— A robot

ACKNOWLEDGMENTS

Thanks all.

ABSTRACT

This is just a placeholder and memorandum
List of things I did

- Part I: modelling and code generation
  - AADL based description of robotic architectures
  - Specialized description of ROS-based components
  - Reusable models of ROS-based design patterns
  - ASN.1 and JSON based description of messages and parameters
  - Engineered reference node for ROS
    - Clear separation between middleware and implementation code
    - Support for internal state machine (node life cycle)
    - Support for external notification of state
    - Encapsulated parametrization
    - Encapsulated internal state
  - Automatic code generation from the AADL ASN.1/JSON model to the reference ROS node
  - Applications: full wheelchair architecture, local planner and global planner
- Part II: abstraction and capabilities
  - Ontology based description of middlewares
  - Specialization of the description on ROS
  - Mapping between ROS messages and general robot capabilities
  - Python based API to interact with the robot using capabilities

  - JSON based decoupling between ROS and an external interface
  - Web based interface with the robot exploiting capabilities

- Part III: considerations
  - Binding between the model and the capabilities
  - Dependencies between capabilities
  - Architecture check on the functionalities of the robot

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

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, i. e., 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 dif-

ferent 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 dangerous 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, i. e., 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

I am not yet sure about the outline of the thesis (i. e., number of parts and chapters). There is a list in the abstract that represent more of less the content of the thesis. The rough idea is to divide it in three or four part. I will have, for sure, one part with the background and state of the art. Possible content of Part I:

- Software engineering
  - Component-Based Software Engineering
  - Software Product Lines
  - Model driven software development
  - Automatic code generation

- System Design and Modelling
  - General Purpose Modelling Languages
    - UML
    - AADL
  - Data modelling languages
    - ASN.1
    - JSON and JSON-schema
  - Ontologies
- Domain specific approaches
  - Automotive
  - Smartphones
  - Space
- Software development in robotics
  - Middlewares and frameworks
  - Robot Operating System
  - Development tools
  - Code generation
  - Best practices and model based approaches

Part I

MODELLING AND CODE GENERATION

2 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 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, an introduction to the Architecture Analysis & Design Language (AADL), a description of the concepts behind the language and how it can be used to model complex system. Then, how the language is exploited to model robotic system and more in particular ROS-based architectures. 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

2.1 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 2.1) 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.

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 2.1: Component categories.

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 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 descrip-

tion 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 characteristics 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 cat-

egory, 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 2.2) 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

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

Table 2.2: Inter-port compatibility.

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 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.

2.2 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 middlewares and frameworks sharing similar design principles can be represented easily with a common language. Moreover, 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, it is regulated by the language itself.

As described in the previous section, 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 middlewares, 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 the size of a memory, the computational power of a processor, the resources used by a process, or the throughput of a connection. 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 the propagation of an error, 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 middlewares. 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 middlewares 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 managed to create a suitable code generator that goes from an AADL model to a complete ROS architecture.

COMPONENT-AND-CONNECTOR STRUCTURE

In robotics, the most popular middlewares and frameworks are based on a *component-and-connector* paradigm, while different approaches implements 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 that are processing 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, again 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 (send and query) and two asynchronous based on a publisher/subscriber paradigm (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 simply and characterise their interactions by the use of interfaces; the result is a *component-and-connector* paradigm. 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., systems, process, device,

subprogram, etc.) support some form of feature (i. e., ports and accesses) to communicate and interact with other components.

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 of the robotic system and it is compatible, at least at an higher level of specification, with multiple middlewares and frameworks. To do so, we defined some common design approaches that are 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 of an architecture. To better define an architecture without including technological details (i. e., specify a middleware or framework), we have to define a general framework 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 exists (and co-exists) 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 computational 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 process and consume 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 implemented by component controlling actuators, they receive commands from

other components and consume them to control the physical device. Alternatively, it is used by storage manager or logger, that 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 process or relay them to the create an output. By going more in details of the internal functioning of this behaviour, it can be divided in two categories: without or with memory.

In the former, the component does not store in any way the data received, they are processed and 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, later, in the main loop or in a different callback, they are processed and circulated back in the computational graph.

REACTIVE – A component has a *reactive* behaviour when its functionalities are synchronously triggered by an message or 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 sys-

tem, 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).

ROBOT OPERATING SYSTEM

Lorem ipsum

ROS TEMPLATES

Could be an independent subsection or part of the previous one

2.3 DATA MODELLING

Lorem ipsum

ASN.1

Lorem ipsum

JSON

Lorem ipsum

JSON SCHEMA

Lorem ipsum

3 CODE GENERATION

Some general introduction on why automatic code generation is important

3.1 FROM AADL TO ROS

3.2 ROS REFERENCE NODE