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 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 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 whit 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
data	Abstraction for data units.
thread	A schedulable execution path.
thread group	An abstraction to logically organise threads.
process	Execution unit with a protected address space.
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 set. 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 the 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 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 artefact, 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 – They represent 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.

THREAD – Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin scelerisque semper mi, a semper tellus condimentum eget. Maecenas rutrum ut odio et efficitur. Phasellus tincidunt lobortis augue eget aliquam. Integer ligula nibh, euismod tempus ullamcorper in, euismod sed lorem. Ut viverra tincidunt dapibus. Sed commodo nibh egestas leo mattis semper in a nulla. Sed porta dictum interdum.

THREAD GROUP – Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin scelerisque semper mi, a semper tellus condimentum eget. Maecenas rutrum ut odio et efficitur. Phasellus tincidunt lobortis augue eget aliquam. Integer ligula nibh, euismod tempus ullamcorper in, euismod sed lorem. Ut viverra tincidunt dapibus. Sed commodo nibh egestas leo mattis semper in a nulla. Sed porta dictum interdum.

HARDWARE COMPONENTS

Lorem ipsum

PROPERTIES

Lorem ipsum

2.2 AADL FOR ROBOTICS

General introduction on why AADL is suitable for robotics

COMPONENT-AND-CONNECTOR STRUCTURE

Lorem ipsum

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