Variability Modeling and Resolution in Component-based Robotics Systems

Luca Gherardi

Thesis

Submitted to the Università degli Studi di Bergamo

for the degree of

Doctor of Philosophy

Advisor: Prof. Davide Brugali

Department of Engineering

Università degli Studi di Bergamo, Italy

Co-Advisor: Prof. Herman Bruyninckx

Department of Mechanical Engineering

Katholieke Universiteit Leuven, Belgium

Ph.D. course in Mechatronics, Information Technology,
New Technologies and Mathematical Methods

XXV Cycle

February 2013

To my mother, my father and my sister

Abstract

The routine use of existing solutions in the development of new systems is a key attribute of every mature engineering discipline. Software reuse is indeed the state of the practice in various application domains, such as telecommunications, enterprise resource planning, automotive and avionics.

One of the key factors that enable the development of reusable software is the flexibility, which is the ease with which a system or a component can be modified to be used in applications or environments that are different from those for which it was originally designed.

In robotics software reuse is still at an early stage. This is mainly due to the complexity and the huge variability that characterize this particular domain. The complexity makes the development of reusable software a task that requires advanced software engineering techniques, which are not always mastered by robotics experts. The variability of hardware, environment and task instead makes frequent and quick the changes that occur in the application requirements.

The research documented in this thesis investigates new approaches for the development of component-based robotics systems, which are flexible enough to accommodate the changes that are likely to occur to the software requirements. Addressing the *flexibility* of robotics software systems and *variability* modeling and resolution are thus the main topics of this document.

The first contribution of this thesis is a software development process that

explicitly takes into account the variability. The process is based on two of the most recent and promising approaches to software reuse, namely the Software Product Lines (SPL) and the Model Driven Engineering (MDE). This thesis describes the process, illustrates a set of models and tools that have been developed for supporting it, and exemplifies its application by means of the robust navigation case study.

The second contribution instead consists of a set of tools and approaches that have been designed in order to develop component-based system with an high level of flexibility and reusability. In particular this approaches focus on addressing the software framework variability and the hardware variability.

Acknowledgments

I'm pretty sure that it would not have been possible to complete the research and the work documented in this thesis without the support and the help of the following people. I really appreciated that and I would like to express my deepest gratitude.

I am tremendously grateful to my mother, my father and my sister. Thank you for taking care of me, and thank you for giving me the opportunity of pursuing my own interests. During the last years I spent most of my time in the office or abroad and I've been only few hours at home. However I've always felt your love.

I would like to thank my advisor Prof. Davide Brugali for giving me the chance of working with him in the context of an European Project. In these three years I learned a lot and I had an incredible number of opportunities to improve my professional skills, enrich my culture and travel.

I would like to thank the Prof. Herman Bruyninckx for giving me the possibility of spending three months in Leuven and working with his group.

I am also grateful to all of my friends, especially the ones who shared with me unforgettable experiences: Alessandra and Davide, Andrea, Carlo and Giorgia, Cristina, Davide and Marta, Marcello, Matteo, don Michele, Nicola, Silvia, Tommy and Verdiana. Thank you for your support, the discussions, the jokes and all those little things that allowed me to take a break during the tiring moments.

I would like to thank my relatives, especially my grandmother, for having been present and interested in the progresses and achievements of my university career.

I would like to thank Andrea and Aldo. We spent a lot of time working together and it has been a pleasure for me. I wish you all the best in both your personal lives and professional careers.

Finally I would like to acknowledge the helpful and important work of the coauthors of the papers I published during my PhD and of the colleagues of the BRICS Component Model Task Force.

Osio Sotto, February 26, 2013.
Luca

Contents

List of Figures

1 Introduction

Recent advances in robotics and cognitive sciences have stimulated expectations for emergence of a new generation of robotic devices that interact and cooperate with people in ordinary human environments.

Simple robotic devices for tasks such as cleaning floors and cutting the grass have met with growing commercial success thanks to their low cost and single purpose design. At the same time, more sophisticated robotic devices such as DLR Justin [1], Care-O-bot 3 [2] and Willow Garage PR2 [3] have been developed for more advanced tasks, such as housekeeping and elder care, further raising expectations but without meeting yet a corresponding commercial success. Versatility and complexity are their distinguishing factors.

Complex means that robots are built by integrating an increasingly larger body of heterogeneous resources. Robotic resources here means the whole set of hardware, software, physical, and virtual entities of limited availability that the robot requires and must manage appropriately in order to accomplish its tasks. These include (a) computation and communication resources, (b) hardware devices, subsystems, and systems, (c) robot functionality, and (d) the physical environment.

Even a simple robotic application, like moving a wheeled robot from place A to place B in an indoor environment, requires several capabilities, such as (1) *sensing* the environments in order to avoid unexpected obstacles (i.e. moving people), (2) *planning* a path from A to B taking into account several constraints (e.g. energy consumption), (3) *controlling* the actuators in order

to execute the computed path correctly (i.e. with a given accuracy), and (4) *reasoning* about alternative courses of actions (e.g. waiting for a passage to get clear or plan a different path).

Sensing, planning, controlling, and reasoning, are human-like capabilities that can be artificially replicated in an autonomous robot as software systems, which implement data structures and algorithms devised on a large spectrum of theories, from probability theory, mechanics, and control theory to ethology, economy, and cognitive sciences. Software plays a key role in the development of robotic systems, as it is the medium to embody intelligence in the machine.

Computing infrastructures of mobile robots have recently evolved from single processor systems to networks of microcontrollers, sensors and actuators, introducing great flexibility in robot capabilities construction. This enables the development of complex robotic systems with hardware/software building blocks that are designed to optimally implement specific functions but whose design is not specific of the robot that integrates them. An example of such building block is the Bluebotics localization subsystem [4]. The number of open source libraries that provide robotics functionalities is growing exponentially thanks to the federated repository and decentralized development approach promoted by the Robot Operating System (ROS) initiative [5].

The new trend, which started only recently, is toward a novel robotic systems engineering approach that fosters an improved reusability of robotic hardware and software components, and promotes a new market, where complex and powerful systems are built by *customizing* and *integrating* reusable building blocks. One of the key attributes of a mature engineering discipline is indeed the routine reuse of existing solutions in the development of new systems.

System *openness* and *flexibility* are key factors that enable the development of reusable software. If a system is flexible, its functionality can be customized by replacing individual components. If a system is open, individual research groups can contribute to its evolution by providing leading edge research solutions.

State-of-the-practice in robotic software reuse however is still at an early stage. This may be due to cultural factors (robotics has traditionally been

the realm of experts in mechanics, electronics, automatic control, computer vision, and artificial intelligence), to contingent factors (so far there has been no sustained push to design reusable and interoperable software), but also to technological and scientific challenges due to the *complexity* and the *variability*.

- *Complexity.* Software for advanced robotics systems is typically embedded, concurrent, real-time, distributed, and data intensive. In addition, robotics software must exhibit specific system properties, such as safety, reliability, and fault tolerance. Developing modular and reusable software components, systems, and applications demands for advanced technical skills both in software and system engineering. Advanced concepts such as software flexibility, portability, scalability, and interoperability must be adequately mastered.

- *Variability.* Robot systems are highly change-centric systems. Robotics is an experimental science that can be analyzed from a double perspective. On one hand, it is a discipline that has its roots in mechanics, electronics, computer science and the cognitive sciences. In this regard, software plays the role of integrator, implementing and bringing together advanced research results in order to build complex robotic systems. On the other hand, Robotics is a research field that pursues ambitious goals, such as the study of intelligent behavior in artificial systems. As a consequence, reusable robotic software artifacts need to be flexible enough to capture quickly-changing technological and functional requirements. More specifically, robot hardware variability, environmental variability, and task variability are major barriers to software reuse. Addressing this challenge demands for a deep knowledge and understanding of the application domain, both in terms of core aspects (entities, functionality, properties) of the domain that are unlikely to change, since they are part of the essence of the domain, and in terms of technological and functional requirements evolution trends.

Despite the severe challenges that researchers and developers have to face in order to introduce software reuse practice in robotic system development,

robotic software reuse is both technologically feasible and economically advantageous. More strongly, software reuse is mandatory in order to make robot software development sustainable in a fast evolving market, where recent advances in robotics and mechatronics technologies have stimulated expectations for emergence of a new generation of robotic devices that interact and cooperate with people in ordinary human environments.

In this context, the research performed during my PhD investigated new approaches for the development of component-based robotics systems, which are flexible enough to accommodate the changes that are likely to occur due to the huge variability in terms of hardware, environment and task. Addressing the *flexibility* of robotics software systems and *variability* modeling and resolution are thus the main topics of this thesis, which will be discussed in the next chapters.

1.1 Thesis structure

The next chapter introduces the concepts of flexibility and analyzes the variability that characterizes the robotics domain. In particular it describes how the variability of hardware, environment, task and software framework influences the design of robotics software systems by taking as example the robust navigation.

The rest of the document is organized in two parts. Part I describes how the concepts coming from two of the most recent and promising approaches to software reuse, namely the Software Product Lines [6] and the Model Driven Engineering [7], have been applied for modeling and resolving the variability in component based Robotics systems.

- Chapter 3 introduces the reuse oriented development process that was defined for developing flexible and reusable component-based Robotics product lines.

- Chapter 4 presents the meta-models and the tools designed for supporting the development process. In particular a set of software framework

independent Meta-models have been defined, which can be easily extended for taking into account Software Framework specific concepts. The chapter introduces the concrete meta-models for two of the most spread Robotics software frameworks (i.e. ROS and Orocos) and for SCA.

- Chapter 5 deepens one of the stages of the Software development process, namely the refactoring, and presents a set of guidelines that describe how to refactor existing open source software libraries in order to facilitate their encapsulation in reusable components. The chapter also illustrates, by means of a case study, how the theoretical guidelines have been applied for analyzing open source implementations of best practice libraries for motion planning and for refactoring one of them. Part of the result presented in this chapter are also documented in [8].

- Chapter 6 describes how the development process previously presented has been applied to the Robust Navigation domain for defining a new Product Line, which allows the modeling and the resolution of the Robust Navigation variability.

Part II describes a set of tools and approaches that have been designed in order to develop component based systems with a high level of flexibility and reusability. Chapters 7, 8 and 9 focus on addressing the software framework variability and the programming language variability. Chapter 10 focuses instead on the robot embodiment variability.

- Chapter 7 faces the problem of integrating the component based robotics software frameworks with the possibility of accessing the World Wide Web (for example for retrieving useful information such as map of the environments, 3D models of furniture or images of objects commonly available at home). This possibility is typically provided by the Service Oriented Architectures (SOA), which are widely spread in the domain of the web based applications. In this direction a Java library that allows the cooperation of SCA components and Orocos components have been

designed and is presented. The results presented in this chapter are also documented in [9].

- Chapter 8 presents a study on the comparison of performance between Java and C++. The goal is to quantify the differences and to offer a set of data in order to better understand whether the performance of Java allows to consider it a valid alternative to C++ or not, at least for non-real-time functionalities. The results of this study support the idea of building hybrid systems, similar to the ones promoted in chapter 7, where C++ is used for real-time functionalities and Java for non-real-time functionalities. The results presented in this chapter are also documented in [10].

- Chapter 9 faces the problem of decoupling the Computation and Coordination concerns during the design and development of component based robotics systems. In particular two different frameworks are used for orthogonally modeling the two concerns: the Service Component Architecture (for the computation) and the Abstract State Machine (for the coordination). The result, presented by means of a case study, is the design of a system in which the functionality implementation and the coordination policy can be changed independently (i.e. it is possible to change the coordination policy without modifying the functionality implementation, and vice versa). The results presented in this chapter are also documented in [11].

- Chapter 10 describes how the Model Driven Approach can be applied to the representation of the kinematics and dynamic constraints of mobile robots. These constraints can be modeled by means of differential equations and are typically used for simulation and sampling based path-planning algorithms. Thanks to the MDE approach the robot differential model is described in a document written according to a specific Domain Specific Language (DCML). This document can be automatically transformed in the source code that implements the differential equations and finally integrated with the simulator or the

path planner. In this way the algorithm code is not hard-coupled to the code that implements the differential model of the robot and, as a consequence, it is more flexible and reusable. The results presented in this chapter are also documented in [12].

Finally chapter 11 draws the relevant conclusions.

1.2 Acknowledgments

The research leading to the results documented in this thesis was supervised by the Prof. Davide Brugali (University of Bergamo) and has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

The author would like to thank all the partners of the BRICS project for their valuable comments.

2

Flexibility and Variability in Robotics Systems

In the software engineering context a software architecture is typically defined as the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them [13]. Components are units of implementation and represent a code-based way of considering the system. Thus, the robot software architecture describes the decomposition of the robot control system into a collection of software components, the encapsulation of functionality and control activities into components, and the flow of data and control information among components. The design or selection of the software architecture specifically takes into account non-functional requirements of a robotic software system (maintainability, portability, interoperability, scalability), that is, those requirements that characterize software quality and enable software reuse.

Ideally, components embedding common robot functionality should be reusable in different robot control systems and application scenarios, and thus they should not be bounded to specific robotic hardware, software-development technologies, or control paradigms. For example, a fully reusable component implementing a mobile robot navigation algorithm should be designed without implicit assumptions about the computational environment (e.g., stand-alone application or distributed system), the possible use (e.g., map building or object tracking), and the robot mechanics (e.g., kinematic model).

In reality, designing reusable components, which can be composed for building flexible systems, consists in finding the best trade-off between being too specific (less reusable) and too generic (less valuable). Three aspects of a reusable component are equally important:

- *Quality.* The quality of a robotics component typically regards the performance and the reliability of the functionality that it encapsulates.

- *Functional reusability.* Typically a software architecture is made of two types of components: horizontal and vertical. Horizontal components provide domain-independent functionalities such as hardware drivers or communication service. Vertical component provides instead domain-specific functionalities such as motion planning or kinematics. Vertical components contribute up to 65% to software reuse while horizontal components no more than 20% [14].

- *Technical reusability.* Technical reusability is mainly concerned with the components' degree of openness and flexibility. A software component can be considered open if its specifications are public, it provides well defined interfaces that promote interoperability[1] with third-part components, and it is portable[2] among multivendor equipment. While the definition of openness is simple, the definitions of component flexibility and system flexibility deserve instead more attention and will be addressed in the next section.

2.1 Flexibility

Flexibility is a concept that has different meaning in different disciplines. Intrinsic to the notion of flexibility is the ability or potential to change and

[1]The IEEE Standard Glossary of Software Engineering Terminology defines interoperability as the "ability of two or more systems or components to exchange information and to use the information that has been exchanged" [15].

[2]The IEEE Standard Glossary of Software Engineering Terminology defines portability as "the ease with which a system or component can be transferred from one hardware or software environment to another" [15].

adapt to a range of states. The common ground on which all disciplines agree is that flexibility is needed in order to cope with uncertainty and change, and that it implies an ease of modification and an absence of irreversible or rigid commitments.

The IEEE Standard Glossary of Software Engineering Terminology defines flexibility as the "ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed" [15].

The term environment refers to the complete range of elements in an installation that interact with the component-based software system. This includes the computer and network platform, the controlled robotic hardware, and the robotic applications that integrate the reusable components. More specifically, flexibility is concerned with the portability of software control system on different robotic platforms (e.g. from a car-like to a omnidirectional wheeled robot), the interoperability among independently developed components (e.g. components interfacing heterogeneous robotic devices), and the reusability of individual components in different application contexts (e.g. a motion planner for static or dynamic environments).

Flexibility of software artifacts depends upon the type of the artifact, the development environment, and the evolution of the artifact's requirements.

- Flexibility is related to the type of software artifact in the sense that properties belonging to the source code, the architecture and the technology used can improve or degrade it. The source code and architecture of the program should be comprehensible and they should make possible to implement changes with easiness.

- The development environment also affects flexibility. One of the most effective ways of dealing with changing requirements is to adopt an agile process for the development of new software. The team should share knowledge regarding the product that must be maintained so that it will be easier to understand which parts of the program should be modified. Lastly the experience of individuals and the quality of every person in the team, as well as the quality of the team as a whole, play

a central role in reducing the mean time to implement a change request in the program.

- Finally, flexibility depends upon the type of changes in the requirements, which typically imply modification to the design of the software components and the system architecture.

While the first and the second points are more related to the developers and their discipline, the system requirements depends only on "external" factors, which cannot be directly controlled (they are indeed typically specified by a customer and also depends on the environment and the available hardware). Moreover, due to the intrinsic change-centric nature of robotic applications, functional and non-functional requirements are characterized by frequently changes.

The most efficient way for addressing the flexibility is therefore being able to predict the class of changes that are likely to occur in the requirements over the lifespan of robotic software components. These changes affect indeed the portability, reusability and interoperability of software components and systems. Being able of anticipating them allows the design of a system architecture that can be easily adapted and reconfigured for supporting the new requirements.

The capability of predicting the requirement changes can be achieve by performing a stability analysis of the domain, which aims to identify the aspects of the domain that are likely to remain stable over the time and separate them from the aspects that are likely to change due to the variability that characterizes the domain. The core of the designed software should be consequently based on these stable concepts, in such a way to be reusable in different applications. Modifying the software for accommodating new requirements will therefore consist in modifying its periphery (variable part) and not its core part.

In conclusion, despite the variability of a domain makes harder the maintenance of software systems in front of new requirements, it is not a drawback. Conversely it has to be analyzed and exploited in order to design flexible

components and systems architectures, which can be easily reused for different applications with different requirements.

2.2 Variability in robotics

A large variety of today's robotic systems is designed according to a small number of robot control architecture paradigms, the most common being the *Sense-Plan-Act* paradigm and the *Layered-Control* paradigm (see [16] for a survey.) According to the *Sense-Plan-Act* paradigm, robot control is the process of taking information about the environment, through the robot's sensors, processing it as necessary in order to make decisions about how to act, and then executing those actions in the environment [17]. The *Layered-Control* paradigm [16] prescribes a clear separation of robot functionalities according to their time scale: the decision layer combines task planning, scheduling, and coordination capabilities; the functional layer provides the typical functionalities for perception, environment representation, localization, navigation, manipulation; the reactive layer implements the low level feedback control loops for motor control, obstacle avoidance, trajectory following.

Despite the architectural similarities, the software that implements robot control applications differs significantly from system to system. These differences relate, for example, to the data structures defined to store relevant information (e.g. the map of the environment, the robot kinematic model), the application programming interfaces (APIs) to drive sensors and actuators [18] and the information model used to represent key concepts (e.g. geometric relations and coordinate representations). These differences are basically due to the huge variability in robot technology and system requirements.

A milestone paper of Rodney Brooks [19] identifies a set of properties of every robotic system, among which three are of interest for this thesis as they represent three variability dimensions that affect the design and implementation of robot control systems, namely *situatedness*, *embodiment*, *intelligence*. An additional dimension in robot variability is represented by the set of software frameworks that have been specifically designed during the last decade for developing robot control systems.

The next subsections illustrate the four dimensions of robotic variability taking the functionalities for the navigation of robots as an example.

2.2.1 Robot situatedness variability

Robot situatedness refers to the fact of existing in a complex, dynamic, and unstructured environment that strongly affects the robot behavior. For example, the environment is a museum full of people where a mobile robot guides tourists and illustrates masterworks, or a game field where two robot teams play soccer, or a manufacturing work cell where a mobile manipulator transports work pieces.

Situatedness implies that the robot is aware of its own posture, in one place at a given time. According to the operational environment, the robot can use different sensors and techniques for 3D perception and localization. For example, a GPS cannot be used inside a building and outdoor it can provide only rough estimate of the robot pose. A stereoscopic vision system can provide accurate 3D information about the surrounding environment but is highly sensitive of the environmental lighting conditions. A laser rangefinder is also highly accurate but cannot detect transparent surfaces, such as a sliding glass door. When possible, the environment could be structured in such a way that the robot can localize itself easily, e.g. by placing visual landmarks in known positions.

Situatedness also implies that the robot can detect and represent the posture of the surrounding objects (walls, furniture, people, etc.). According to the environment characteristics (e.g. highly cluttered, open spaces, long corridors, etc.) and to the robot task, different types of map can be used to represent the operational environment. A continuous geometric representation is adequate to represent the floor of a building with rooms and corridors, while a topological representation is adequate to represent the streets of a city or the hallways of a large building. If the environment is dynamic, the robot should be able to represent static and moving objects explicitly. A hybrid topological-geometric map could represent the positions of visual landmarks and the pathways that connect them.

Robot control applications strongly depend on the type of sensors, map, and environment, since different algorithms are used to process sensory information (e.g. filtering, interpreting, fusing), to update the map of the environment, to localize the robot with respect to the map, and to plan an obstacle-free path.

2.2.2 Robot embodiment variability

Robot embodiment refers to the consciousness of having a body (a mechanical structure with sensors and actuators) that allows the robot to experience and interact with the world. The robot receives stimuli from the external world and executes actions that cause changes in the world state. Simulated robots may be "situated" in a virtual environment, but they are certainly not embodied.

Sensors allow the robot to perceive the environment's changes and to react to them by changing its own behavior (e.g. the detection of a very closed obstacle causes the robot to switch from the operating to the emergency mode). By means of its sensors the robot experiences the effect of its actions on the environment through the effects that such actions produce.

With increasing computational power made available by advances in microelectronic technology, smart sensors and actuators integrate sensing, actuating, processing, and networking elements into a single device. They improve the modularity of the computing infrastructure by locally performing some signal-processing tasks but at the same time increase the complexity of software applications, which become highly distributed and decentralized.

Despite the semantic similarities between the operations supported by similar devices (e.g. all ranging devices provide distance measurements, all rovers provide wheeled mobility), the externally visible behavior of the software that abstracts and interfaces to each device greatly depends on the device hardware architecture [18]. There are devices that have their analog and digital signals directly mapped to memory registers on the central processor. For these devices, all basic functionality (e.g. measurements filtering, image processing, pulse-width modulation (PWM) motor control, camera synchronization, etc.)

are implemented in software, which thus requires dedicated computational and synchronization resources. On the other hand, there are devices that implement much of their low-level functionality in their firmware and thus reduce the load on the central processor.

Furthermore, the robot mechanical structure greatly influences the implementation of individual functionalities and even the architecture of the software application that controls the robot. For example, the control algorithm that drives the robot along the computed trajectory depends on the kinematics model of the rover (i.e. a differential drive or an omnidirectional rover). As another example, a video camera can be attached to the wrist of a manipulator arm mounted on the rover. In order to interpret the camera images correctly (i.e. for rover localization), the position of the camera should be tracked while the arm is moving. This functionality is not required if the camera is mounted on the rover in a fixed position.

2.2.3 Robot intelligence variability

Robot intelligence refers to the ability to express adequate and useful behaviors while interacting with the dynamic environment. Intelligence is perceived as "what humans do, pretty much all the time" [19]. The concept of intelligence (the ability of expressing useful behavior) is quite elusive. It is usually associated to other concepts, such as: autonomy, i.e. the robot's ability to control its own activities and to carry on tasks without the intervention of the human operator; deliberativeness, i.e. the ability of planning and revising future actions in order to achieve a given goal while taking into account the mutable conditions of the external environment; adaptability, i.e. the ability of changing its behavior in response to external stimuli according to past interactions with the real world.

In complex robot control applications, several activities interact with the hardware devices and with each other concurrently. For example, the robot uses a 3D depth camera for both map building and for obstacle avoidance, controls both the base and the arm of a mobile manipulator for object grasping, control the wheel motors to follow a path while it is replanning the

path. Typically, low-level control loops regulating the robot motion are better implemented according to the synchronous data flow model of computation, where the control software periodically requests new measurements from sensors (e.g. encoders) and sends motion commands to the actuators. Higher-level activities (such as motion planning and task planning) interact mostly according to the asynchronous event-based model of computation in order to react promptly to changing conditions in the operational environment (e.g. an unexpected closed door).

In Robotics, the coordination of concurrent activities is typically control-driven and modeled using state-charts and finite states machines, which define the state of the computation of the entire system at any moment in time in terms of the current internal state of each component. Components observe state transitions in the systems by listening to events notified by other components.

2.2.4 Software frameworks variability

During the last few years, many ideas from software engineering (such as component-based development and model-driven engineering) have been progressively introduced in the construction of robotic software systems, in order to simplify their development and improve their quality (see [20] for a survey).

Compared to more traditional business applications, in Robotics the software developer faces the complexity of event-based, reactive, and distributed interactions between sensors and motors and between several processing algorithms. Managing concurrent access to shared resources by multiple (distributed) activities is one of the main issues, as thoroughly discussed in [21]. For this reason, robotic-specific component-based frameworks and toolkits have been developed (ROS, Orocos, OpenRTM, Opros, Smartsoft, Yarp and many more), which offer mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management, and system configuration.

This subsection introduces two of the most spread robotics software

framework: ROS and Orocos. Moreover one of the most used software frameworks in the context of the Service Oriented Architectures, namely SCA, is presented. Despite SCA is not robotics specific, it will be used several times along the chapters of this thesis.

All these software frameworks share two characteristics, which are however implemented in a different way (they are further discussed in the part I of this document).

- They provide a component model, which defines a set of architectural elements (e.g. components, interfaces, connection) and the rules for composing them in order to build a component based systems. These component models, except in the case of SCA, are typically not explicitly modeled (i.e. they are not formalized according to the principles of the Model Driven Engineering), and, despite they provide similar concepts, they differ syntactically.

- They provide a runtime infrastructure (in this document it is called *deployer* or *runtime*), which is in charge of instantiating, connecting, configuring and activating the components that are part of the system. The deployer executes these operations according to a set of instructions that are typically defined in an XML file (in this document it is called *deployment file*).

The rest of this subsection provides a brief overview of ROS, Orocos and SCA.

The Robot Operating System

The Robot Operating System (*ROS*)[22] is a message-based peer-to-peer communication infrastructure supporting the easy integration of independently developed software components, called ROS nodes. A ROS system is thus a computation graph consisting of a set of nodes communicating with each other. Nodes are blocks of functional code and are implemented as classes (typically in C++ or Python) that wrap robotic software libraries and provide access to the communication mechanisms of the underlying infrastructure (the ROS

core). Messages are typed data structures that can be nested into compound messages and are exchanged between nodes according to the publish/subscribe communication paradigm in an asynchronous manner without the need for the interacting nodes to know each other and to participate to the interaction at the same time. ROS is currently the most popular robotic framework, due to the toolchain support (tools for compiling, deploying, debugging and plotting) and to the huge amount of open source libraries packaged as ROS nodes.

The Open RObot COntrol Software

The Open RObot COntrol Software (*Orocos*)[23] is one of the oldest open source software frameworks in robotics, under development since 2001, and with professional industrial applications using it since about 2005. The focus of Orocos has always been to provide a hard real-time capable component framework, the so-called Real-Time Toolkit (RTT) implemented in C++ and as independent as possible from any communication middleware and operating system. Components interact with each other by exchanging data and events asynchronously through lock-free input/output ports according to the Data Flow communication paradigm. The distinguish feature of OROCOS is the definition of a component model that specifies a standard behavior for concurrent activities. Components with real-time, deterministic and cyclic behavior get fixed and cyclic time budgets for computation and within a computation cycle they must reach stable intermediate states.

The Service Component Architecture

The Service Component Architecture (SCA)[24] defines a model for creating component-based applications that follow the service oriented architecture principles. The first version of its specification was released in March 2007 and was the result of the collaboration of different partners such as IBM and Oracle.

SCA defines a generalized notion of a component, where provided interfaces are called *Services* and required interfaces are called *References*. Services

and references are thus typed by interfaces, which describe sets of related operations that can be invoked synchronously or asynchronously.

The components in a SCA application might be built with Java or other languages, or they might be built using other technologies, such as the Abstract State Machines Language (ASML) (see chapter 9).

SCA is supported by graphical tools, which build on the Eclipse Modeling Framework [25] and allow the generation of a deployment file from a graphical representation of components and systems. ROS and Orocos do not provide a similar tool. However during the BRICS Project an Integrated Developed Environment (BRIDE)[26] was developed, which allows the design of deployment files for ROS and Orocos starting from a graphical representation.

Part I

Variability Modeling and Resolution

3

A Reuse Oriented Development Process

The routine use of existing solutions (i.e. reuse) in the development of new systems is a key attribute of every mature engineering discipline. Software reuse is a state of the practice development approach in various application domains, such as telecommunications, factory automation, automotive, and avionics. Software Engineering has produced several techniques and approaches for promoting the reuse of software in the development of complex software systems. A survey can be found in [27] (Sidebar *A Historical Overview of Software Reuse*).

In Robotics, software reuse is typically conceived as cut and paste of code lines from program to program: this practice is called opportunistic software reuse and might work only for the development of simple systems (e.g. for educational purposes) or for unique systems (e.g. a research prototype). Most of the best practice software libraries for robotics are based on software architectures invented from scratch each time. Many valuable robotic applications are monolithic systems that have been developed to solve a specific class of problems.

In contrast, the development of industrial-strength robotic systems that aim to become commodity, require a systematic approach to software reuse. Systematic software reuse is the routine use of existing software or software knowledge to construct new software, so that similarities in requirements, architectures and design between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.

If a company that commercializes integrated robotic systems wants to achieve customer value through large commercial diversity with a minimum of technical diversity at minimal cost, the best approach to software development is the Software Product Line (SPL)[6].

An SPL is a set of applications (products) that share many (structural, behavioral, etc.) commonalities and together address a particular domain. The term domain is used to denote or group a set of systems (e.g. mobile robots, humanoid robots) or functional areas (motion planning, deliberative control), within systems, that exhibit similar functionality. Each new application is built from the SPL repository of common software assets (e.g. architectural and design models, software components).

In order to exploits the SPL approach, a new software development process has been defined. This process accounts for two peculiarities of the robotics field:

- Today, a huge corpus of software applications, which implement the entire spectrum of robot functionality, algorithms, and control paradigms, is available in robotic research laboratories and potentially could be reused in many different applications. Typically, their interoperability or their extensions towards novel applications require high efforts. Any company that aims at developing professional software for complex robotic systems has to make an initial investment in refactoring and harmonizing existing open source robotics libraries that implement the robot functionalities offered by the SPL. This phase is typically called *software development for reuse*.

- Typically, robotic systems integrators are not software engineers and do not master advanced software development techniques adequately. For this reason, the proposed development process exploits the Model-Driven Engineering (MDE) [7] approach. According to the MDE approach, robotic system integrators use domain-specific languages to build models that capture the structure, behavior, and relevant properties of their software systems. A new application is developed by reusing these models, customizing them according to specific application requirements,

and semi-automatically transforming models and even generating source code using transformation engines and generators. This phase is typically called *software development with reuse*.

This chapter aims to illustrate the whole software development process that has been defined for developing flexible and reusable component-based robotics product lines.

3.1 The development process

The reuse-oriented development process that has been defined and applied during the BRICS project is made of three phases, which are depicted in figure 3.1. The first two phases, namely *Capabilities Building* and *System Building*, are intended to produce software and models *for reuse*, while the remaining phase, namely *System Deployment*, supports the development of software *with reuse*.

Each box represents stage, which receives as input and produces as output one or more models and/or software libraries. The stages (except *Algorithms development*) are described in details in the following sections while the models will be introduced in the chapter 4.

Vertical dashed lines divide the figure in the three phases of the development process, while horizontal dashed lines separates the figure in three parts that correspond to different roles, namely the *Software Engineering Expert*, the *Robotics Expert* and the *System Integrator*.

- The **Robotics Expert** designs robotics algorithms, which are implemented as *Class Libraries*, and represents the functional variability of the robotic product line by means of a *Feature Model*.

- The **Software Engineering Expert** designs the *Class Libraries* by refactoring the *Legacy Code*, represents the architectural variability in the software product line with a *Template System Model* and maps functional variability to architectural variability with a *Resolution Model*.

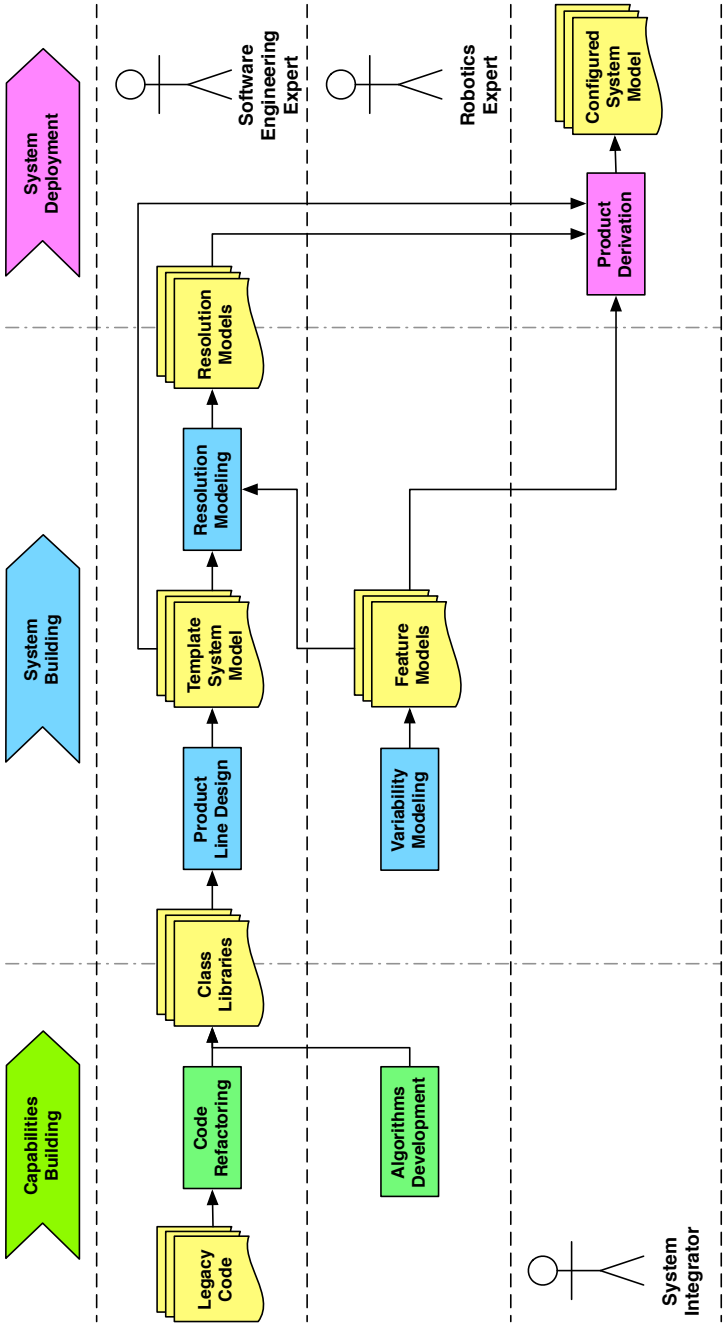

Figure 3.1: The development process

- The **System Integrator**, supported by a specific tool, selects the desired functionalities for his application and generates the *Configured System Model*.

3.2 Code refactoring

Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [28]. It occurs at two complementary levels:

- *Syntactical refactoring* is a behavior preserving transformation that, through the adoption of good design principles (abstraction, information hiding, polymorphism, etc.) aims at making software artifacts modular, reusable, open.
- *Semantic refactoring* is a domain-driven transformation that, through a careful analysis of the application domain (commonality/variability and stability analysis), enhances software artifacts flexibility, adaptability, and portability.

Software refactoring brings many advantages not immediately but in a long time. The initial cost in terms of time and effort spent for rewriting the code is balanced by the time gained in future. This gain is due to a code more readable, more reusable and more maintainable. The result of a refactoring process is a library of classes that are software framework independent, are organized in a hierarchy of abstraction levels, provide harmonized interfaces (API), and implement a variety of algorithms.

The objective of this stage of the development process is the definition of a set of class libraries that can be easily encapsulated into software components, which provide the properties described in the next subsection.

3.2.1 Refactoring towards components

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A

> software component can be deployed independently and is subject to composition by third parties." [29]

Software components come with well-defined *component specifications*, which are abstractions from the details (data structures and operations) of their (possibly many) implementations. A component specification explicitly declares which functionalities (*provided interfaces*) are offered to its clients, the public obligations (*contracts*) with its clients in the form of various kinds of constraints (e.g. pre-conditions, post-conditions, invariants) on how to access the functionalities, and the dependencies (*required interfaces*) to the functionalities that are delegated to other components.

A *component implementation*, on the other hand, defines how the component supports those features and obligations in terms of a collaborative structure of realizing objects (class instances) and algorithms implementing the functionalities declared in the component specification.

Separating the specification of components from their implementation is desirable for achieving modular, interoperable, and extensible software and allows independent evolution of client and provider components. If client code depends only on the interfaces to a component and not on the component's implementation, a different implementation can be substituted without affecting client code. If a coherent set of *required* interfaces can be defined that specify the most frequently used robot services and capabilities, and if robotics applications are designed around those interfaces, then every component implementing compatible *provided* interfaces has the potentiality to be reused in those applications.

The various implementations of a component may differ in functional characteristics (i.e. different algorithms for motion planning), non-functional properties (i.e. performance, maintainability, documentation quality, reliability), realizing technology (e.g. the description of the geometric space may be stored in a relational database or as XML files) and even programming language (if components are build on a middleware or multi-language run-time infrastructure).

Despite of these differences, components that implement the same interfaces and offer similar functionalities are typically implemented around common entities and mechanisms, which are core aspects of the provided functionalities and can be represented as stable data structures and operations. In contrast, those aspects of a component implementation that are more likely to be affected by the evolution of the application domain represent its variation points.

Component frameworks enable a clear separation between stable and variable aspects of a component implementation. A component framework is a skeleton that can be specialized to produce custom components. As such it represents a family of component implementations, which can be derived from its design and built on its data structures and operations without changing them.

Components frameworks can be customized at design time, when the software developer implements specific variants (e.g. algorithms) for each variation point, or at run time, when one of several alternative variants is selected according to current execution context.

In order to transform open source libraries in a set of classes that can be easily wrapped into software components that provides the features described above, a refactoring process, which consist of a set of well-known architecture refactoring patterns have to be applied. These patterns provide concrete guidelines to restructure the architecture of software systems.

Refactoring patterns regards the redistribution of the responsibilities among the classes of a software library, the harmonization of the common data structures and to reduction of the coupling degree. Some of them are mostly related to the definition of components interfaces (e.g. Move Behavior Close to Data or Split up God Class), while other are mostly related to components implementation (e.g. Transform Conditionals into Registration or Eliminate Navigation Code). More details about the refactoring process and the refactoring patterns are described in the chapter 5.

3.3 Product line design

The Product Line Design is the process of designing reusable component frameworks (see subsection 3.3.1), which encapsulate the classes produced by the algorithms development and the code refactoring phases, and Software Product Lines architectures, which are built by assembling the designed components (see subsection 3.3.2). In this stage the *Software Engineering Expert* has to take in account the variability and use software engineering techniques for designing artifacts that can be easily adapted for satisfying new requirements.

3.3.1 Component design

Section 3.2 and [27] have defined a set of architectural principles for the design of Component Frameworks, which specifies two variability mechanisms that are common to most of the component models.

- Separation of a component interface form its implementation and the consequent possibility to replace the implementation without the need to modify the interface.
- Definition of component properties and the possibility to set their values at deployment-time

These principles can be described by means of an Abstract Component Meta-Model, which provides the rules for designing components that are software framework independent and provides the variability mechanisms described above. The Abstract Component Meta-Model defines the following architectural elements:

- System: is the main entity of the Abstract Component Meta-Model and is a composition of component frameworks, whose interfaces are connected by means of Connections.
- Component Framework: is a software package that encapsulates and provides a set of functionalities and relies on another set of functionalities provided by other components (in the next pages of this document

the Component Framework is typically abbreviated as Component). All these functionalities are provided and accessed by means of the component Interface. Each Component has a reference to an *Implementation*, which is the executable or the dynamic library that implements the component provided interfaces. A Component can be customized by selecting different implementations that conform to its interfaces.

- Interface: explicitly defines (in terms of data types and contract) how a component provides (provided interface) or requires (required interface) a service or a data-flow.

- Property: is a mechanism that allows the Component configuration. A property provides an interface for setting the value of a parameter defined in the component implementation (e.g. component period or algorithm parameters).

- Connection: defines the connection between a Provided Interface and a Required Interface. It also specifies the communication policy.

3.3.2 Product line architecture design

Several components can be assembled to build applications. A family of similar applications that are built reusing a set of software components and share the same architecture is called a Software Product Line. The product line architecture specifies the structural commonalities among the applications (stable parts and variation points) and the variations reflected in each application (variants). It prescribes how software components can be assembled to derive individual products. For example, a variation point in a typical robust navigation product line is the algorithm for obstacle avoidance. Different algorithms (i.e. dynamic window approach [30] or vector field histogram [31]) are implemented as distinct software components. The product line architecture guarantees that these components are interchangeable.

In this stage the Software Engineering Expert defines the *Template System Model*, which represents all the possible configurations of a software product line and specifies:

- a set of components that can be used for building all the possible applications of the family (some of them are mandatory, some others are instead optional),
- a set of connections among components (some of them are stable, some others are variable).

By selecting the optional components, their specific implementation, the values of their configuration properties, and the variable connections, the variability of the product line is resolved. In this way the *Template System Model* is transformed in a model that describes the architecture of a specific application, which belongs to the Software Product Line and provides a specific set of functionalities. The new model is called *Configured System Model*. The Configured System Model can be manually defined or can be automatically generated starting from a selection of functionalities, as described in the next sections.

3.4 Variability modeling

Building software systems according to the product line approach is economic and efficient [6]. Most work is about integration, customization, and configuration instead of creation. A system configuration is an arrangement of components and associated options and settings that completely implements a software product. Variants may exclude each others (e.g. the selection of a component implementing an indoor navigation algorithm excludes the choice of components providing GPS-based localization services) or one option may make the integration of a second one a necessity (e.g. a component implementing a visual odometry algorithm depends on a component that supplies images of the environment). Hence, only a subset of all combinations is the admissible configuration.

In order to model and symbolically represent the product line variation points, their variants and the constraints between them, a formalism called Feature Models was introduced in 1990 in the context of the Feature Oriented Domain Analysis (FODA) [32]. While the Software Product Lines describe

the architectural variability, Feature Models represent the variability in term of functionalities.

This stage of development process aims to define Feature Models that make explicit the functional variability that was implicitly defined during the product line design. These models highlight both the variability provided by each single component framework and the variability provided by their composition in software product lines.

3.4.1 Feature models

A feature model is a hierarchical composition of features. A *feature* defines a software property and represents an increment in program functionality. Compose features, or in other words select a subset of all the features contained in a feature model, corresponds to define a possible configuration of a software that belongs to the application domain described by the model. This selection is usually called *instance*.

On the base of the feature models, the FODA experts have defined a graphical representation called *feature diagram*. The description of the Feature Models reported below refers to the feature diagram depicted in figure 3.2 in order to exemplify the characteristics of feature models. The diagram describes the functionality of a motion planning product line.

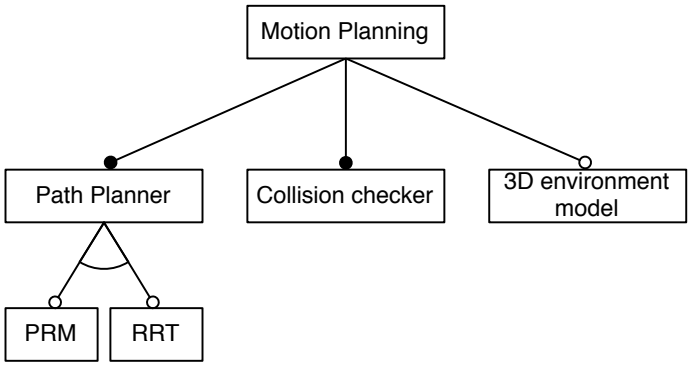

Figure 3.2: The Feature Diagram of a motion planning product line

Feature models are organized as a tree and the root feature, also called

concept, defines the application domain. Features are represented by means of white boxes, which contain the feature names, and are connected to the children features by means of edges, which represent containment relationships. Features can be discerned in two main categories:

- ***Mandatory***. Mandatory features **have to** be present in every possible configuration of a software that belongs to the domain described by the model. They usually define the core of the software and represent functionality or properties that are fundamental in the specific domain: the commonalities. In the feature diagrams they are depicted by means of a black circle on the top. In the example the *Collision Checker* functionality is a mandatory feature.

- ***Optional***. Optional features **can** be present but they are not mandatory. They represent functionality or properties that characterize a specific configuration of the software: the variabilities. In the feature diagrams they are depicted by means of a white circle on the top. In the example the *3D environment model* functionality is an optional feature, which means that not all the motion planning applications provide it.

Features can be connected to their children features by means of two types of containment relationships:

- ***Or*** *containment*. It is a relationship between the parent feature and a set of children features. It means that from the children features **at least one** has to be present in a possible configuration of the software. This relationship is depicted by means of the black semi-circle that connects the edges.

- ***Alternative*** *containment (X-Or)*. It is a relationship between the parent feature and a set of children features. It means that from the children features **only one** can be present in a possible configuration of the software. In the example it is represented by the containment between the *Path Planner* feature and its children. This relationship is depicted by means of the white semi-circle that connects the edges.

The basic feature models also define two kinds of constraints between the features: *requires* and *excludes*. These constraints allow the definition of a subset of valid configurations. They are typically expressed in the form A *kind_of_constraint* B, where A and B can be a simple feature or a composition of features by means of logical operators (AND, OR, XOR, NOT).

- ***Requires*** *constraint.* It means that if a feature A is selected to be part of a configuration, then also a feature B **has to** be selected. If A and/or B represent logical rules the constraint imposes that if A is true, then also B has to be true. To be noticed that for solving the logical rules the value of a feature has to be considered true if the feature is selected.

- ***Excludes*** *constraint.* It means that if a feature A is selected to be part of a configuration, then a feature B **cannot** be selected. If A and/or B represent logical rules the constraint imposes that if A is true, then B has to be false.

3.4.2 Cardinality-based feature models

The cardinality-based feature models propose to replace the properties *optional* and *mandatory* and the containments *or* and *alternative* with a cardinality-based annotation. In particular these ideas are proposed in two different works:

- [33] proposes a ***feature-cardinality*** approach. The idea consists of marking each feature with a lower bound and an upper bound. The upper bound defines the maximum number of times that the feature can be present in an instance.

- [34] proposes a ***containment-cardinality*** approach. The idea consists of marking each containment with a lower bound and an upper bound. The lower bound defines the minimum number of sub-features that have to be present in an instance whereas the upper bound the maximum number. According to this approach the containment relationships "*or*" and "*alternative*" are respectively replaced by $[1 \dots *]$ and $[1 \dots 1]$.

3.4.3 Extended feature models

The extended models propose to attach some information to the features by means of attributes [33]. The purpose of the attributes is to allow a more concise representation of feature models. In fact the idea is to use the attributes for representing information that are important but not so relevant to be represented as features. Attributes are defined by means of a name, a type and a value.

3.5 Resolution modeling

Once the architecture of the product line has be defined and its functional variability modeled, the next phase of the development process regards the definition of how the Template System Model has to be modified for producing a Configured System Model. This information is encoded in the *Resolution Model*, which specifies how each variation point of the *Template System Model* has to be resolved accordingly to the variant selected for the corresponding variation point in the *Feature Model*.

The *Resolution Model* is based on the concepts of *Required Element* and *Transformation*.

A *Required Element* reflects an architectural element that is defined in the Template System Model and that has to be present in the Configured System Model when the associated feature is selected. There are two types of required elements: *Required Components* and *Required Connections*.

A *Transformation* is instead an action that modifies the architectural elements of a Template System Model in order to configure them accordingly to the selected features. According to the concept of Component Framework, different kinds of transformations are available, which are described in the following list.

- ***Implementation Transformation***: this transformation allows the software engineering expert to define which implementation will be used for a certain component, i.e. which algorithm. The software engineer

specifies a link to the desired component (defined in the Template System Model) and the class that implements its interface.

- ***Connection Transformation***: this transformation allows the software engineering expert to define how the components will be connected. The software engineer specifies a set of connections that have to be created.

- ***Property Transformation***: this transformation allows the software engineering expert to define the value of a certain property. The software engineer specifies a link to the desired property (defined in the Template System Model) and the value he wants to assign to it.

The *Software Engineering Expert* can associate to each feature one or more required elements and one or more transformations. A selection of a set of features will hence result in the execution of a set of transformations and the removal of all the elements of the product line model that are not defined as required.

3.6 Product derivation

The last phase of development process regards the deployment of a specific application belonging to the product line and can be achieved by resolving the product line variability.

In this stage the System Integrator can select the set of variants (features), which reflect the functional requirements of his application. This selection should satisfy the explicit constraints, the containments cardinalities and the selection of the mandatory features defined in the feature model. The tool presented in the chapter 4 automatically checks the constraints satisfaction and only successively generates the *Configured System Model*, which can be then transformed in the deployment file of a specific software framework by using a model-to-text transformation.

4

Variability Modeling and Resolution: Models and tools

This chapter presents the meta-models and tools developed in order to support the development process described in the chapter 3. These meta-models and the tools are based on the Eclipse Modeling Framework (EMF) [25] and the Graphical Modeling Framework (GMF) [35], which are two Eclipse projects that allow the development of domain specific languages (DSLs) and graphical editors for their visualization. Every DSL is built on a set of rules that are defined by means of a formal meta-model, which describes the basic elements and how they can be composed for writing a document conforms to the specific DSL. In the Eclipse environment these meta-models are typically expressed through the Ecore format [36], which is a small and simplified subset of UML that will be used in this chapter.

4.1 A model driven approach

Some of the robotic software frameworks (see section 2.2.4 for an overview) are distributed with a tool for design-time modeling and deployment-time configuration of component-based systems. For example, the BRIDE toolchain developed by the BRICS project for Orocos and ROS allows the user to graphically design the system architecture and to generate a configuration file (the *Configured System Model*) that is stored as XML file and can be

transformed in a deployment file (by means of a Model-to-Text transformation). The deployment file is then loaded by the runtime infrastructure of the software framework, which instantiates, connects, configures, and activates the system components.

Every time the robotic system configuration needs to be adapted (because of different embodiment, situatedness, and intelligence requirements), the user (i.e. the system integrator) has to modify the *Configured System Model*. This means that the system integrator should be an expert in both robotics and software engineering in order to understand how the variability in the component system architecture needs to be resolved to accommodate the variations in system requirements.

In order to overcome this issue two Eclipse plugins have been defined (see the *Feature Selector* and the *Resolution Engine* in figure 4.1). These tools help the **robotics expert**, the **software engineering expert** and the **system integrator** in the execution of the phases of the development process introduced in chapter 3.

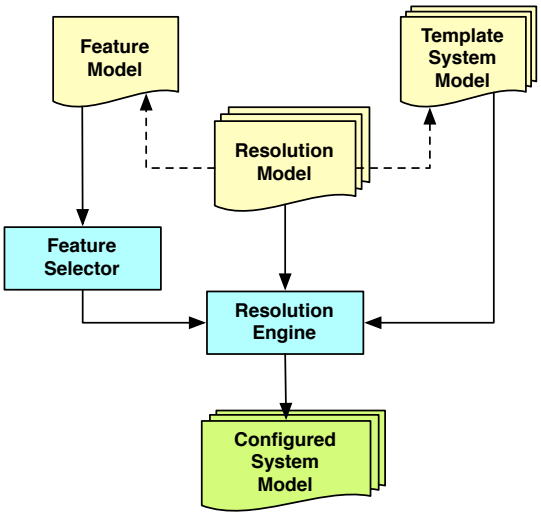

Figure 4.1: Modelling and Resolving Robotics Variability

As described in section 3.3, the *Template System Model* specifies the set of components that can be used for building all the possible system configurations (i.e. product line applications) and a set of stable connections

between components. This model allows the definition of several kinds of architectural variation points: (a) some of the components can be optional, (b) new connections can be created, (c) components can be configured by setting their implementation and the values of their properties.

The resolution of all the variation points of the *Template System Model* produces the *Configured System Model* that describes a specific configuration of the system. For this reason the *Template System Model* can be viewed as a skeleton, which has to be customized in order to produce a specific *Configured System Model*.

The *Resolution Model* specifies how a variation point of the *Template System Model* has to be resolved accordingly to the variant selected for the corresponding variation point in the *Feature Model*.

The *Feature Selector* is a tool, which allows the user to create instances of feature models and verify that all the constraints are satisfied. The *Resolution Engine* receives as input an instance of the *Feature Model*, a collection of *Template System Models* and of *Resolution Models* (a pair for each of the supported component frameworks, i.e. ROS, Orocos and SCA) and produces as output a *Configured System Model*.

Figure 4.2 illustrates the models and meta-models defined for modeling and resolving architecture variability. They are grouped into three orthogonal layers, which are structured in levels of abstraction: M2 is the level of the meta-models, while M1 is the level of the concrete Models. In some cases these levels are further divided in two sub-levels.

- *Product Line Modeling Layer*
  - *M2 Abstract*: the Abstract Component meta-model defines the rules for modeling Component-Based product lines. It is Software-Framework independent, and describes the architectural elements for expressing variability in software architectures.
  - *M2 Concrete*: the Concrete Component meta-models describe the rules for modeling systems accordingly to a specific software framework (e.g. ROS, Orocos or even SCA, which is not reported in order to keep the figure as simple as possible).

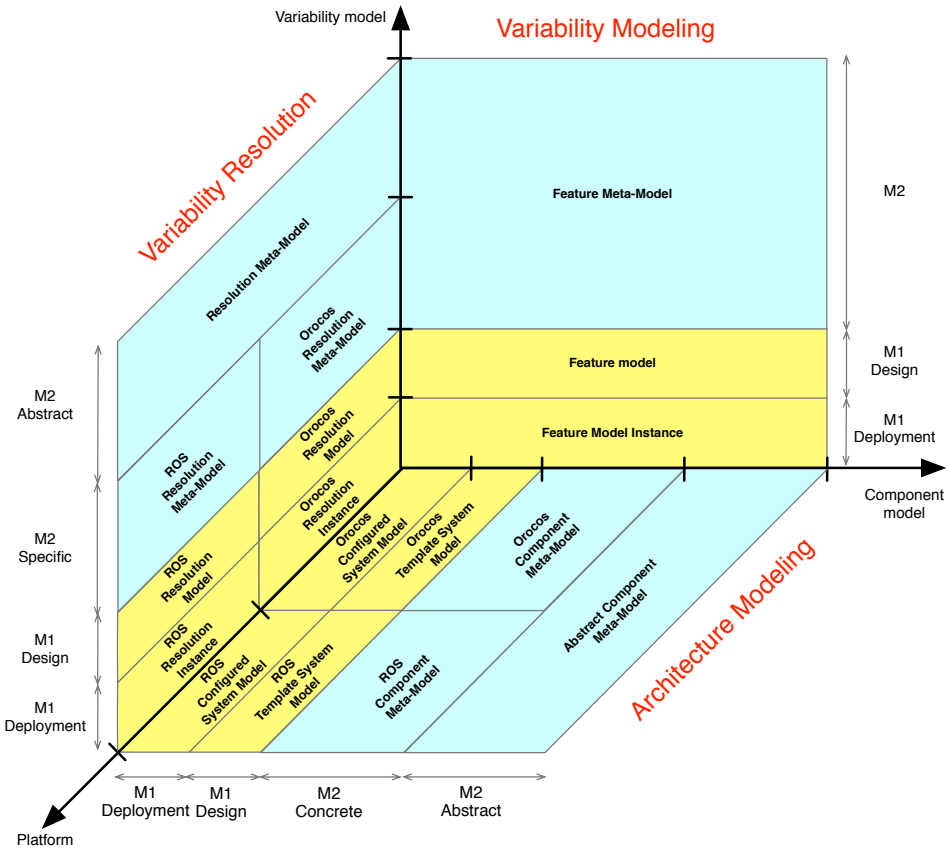

Figure 4.2: Meta-models representation

  - *M1 Design*: the Template System Models are defined for a specific foftware framework.
  - *M1 Deployment*: the Configured System Models define a specific system configuration (i.e. a specific application).

- *Variability Modeling*

  - *M2*: the Feature meta-model defines the rules for creating Feature Models.
  - *M1 Design*: the Feature Model describes the functional variability of a product line in terms of variation points and variants.

  - *M1 Deployment*: an instance of a Feature Model resolves the variability by defining a variant for each variation point.

- *Variability Resolution*

  - *M2 Abstract*: the Resolution meta-model is independent from any software framework and defines the basic concepts for variability resolution.
  - *M2 Specific*: these meta-models extend the Resolution Meta Model with software framework-specific details.
  - *M1 Design*: these models associate a model-to-model transformation to each feature defined in a Feature Model for transforming a Template System Model into a Configured System Model.
  - *M1 Deployment*: these models contain the sub-set of transformations defined in the M1 Design model, which correspond to the selected variants of a Feature Model instance.

The above mentioned meta-models, models and tools are described in details in the following sections.

4.2 Product line modeling

Level M2 defines the *Abstract Component meta-model* and three concrete component meta-models for ROS, Orocos and SCA.

At M1 level, the concrete component meta-models are used to define the *Template System Models* according to the ROS, Orocos or SCA component model.

Table 4.1 summarizes how the variability mechanisms defined in the *Abstract Component meta-model* are mapped to the specific mechanisms provided by ROS, Orocos and SCA.

The next subsections present an UML representation for or each Concrete meta-model and describe how the specific concepts map on the concepts defined in the Abstract meta-model

Abstract CMM	ROS	Orocos	SCA
System	System	Package	Composite
Component	Node	Task Context	Component
Provided Int.	Publisher ServiceServer	Output Port	Service
Required Int.	Subscriber ServiceClient	Input Port	Reference
Connection	Publisher.target Topic Subscriber.source --- ServiceClient.srv Service ServiceServer.srv	Connection	Wire
Property	Parameter	Property	Property

Table 4.1: The mapping between the concepts of the Abstract Component meta-model and the concepts defined in the ROS, Orocos and SCA frameworks

4.2.1 The abstract component meta-model

The Abstract Component meta-model, which was already described in subsection 3.3.1, is depicted in figure 4.3.

It has to be noted that the abstract component model doesn't make any assumption on the architecture Component-and-Connector style (e.g. Data Flow, Event-Based, Client-Server, etc.) [37, chap. 4].

A Template System Model defined by means of the Abstract Component meta-model can be transformed in a new model defined by means of one of the Concrete Component meta-models by executing a Model-To-Model transformation. However this process cannot be completely automated, indeed the Software Engineer has to manually specify a set of information that is specific for the target Software Framework.

4.2.2 The ROS component meta-model

The Component meta-model of ROS is depicted in figure 4.4[1]. A ROS *System* is a graph of *Nodes*, *Topics* and *Services*.

[1]The ROS meta-model is a revised version of the Ecore model available at `https://github.com/abubeck/bride/tree/master/bride_plugin_source/`

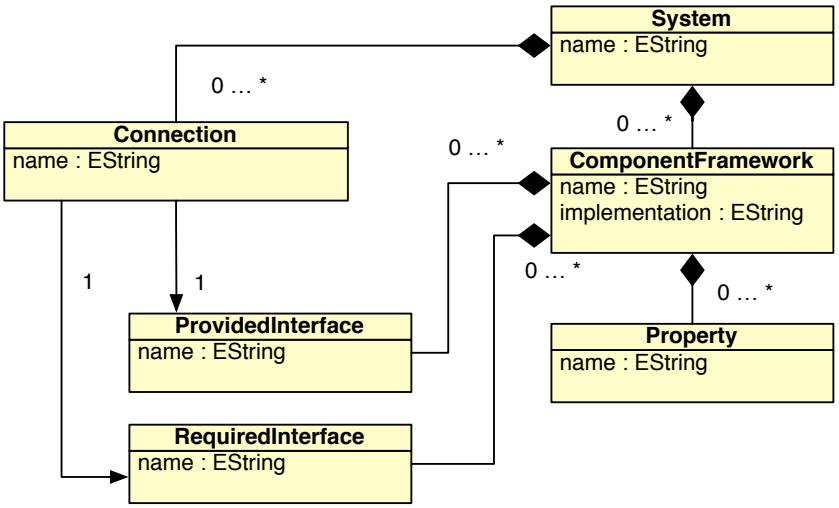

Figure 4.3: The abstract component meta-model

A *Node* is the ROS equivalent of a component and is a stand-alone executable. The reference to its implementation is defined by means of two values: the *package* and the *name*. In ROS a package is a folder containing the implementation of software libraries and nodes, which are characterized by their name. The couple package-name has to be unique in the ROS file system and identify the implementation of a node. Changing these values allows us to change the node implementation. Nodes provide four kinds of interfaces: *Publishers*, *Subscribers*, *Service Servers* and *Service Client*.

Publisher and Subscriber are interfaces (respectively provided and required) that can be used for implementing a communication based on the Publish-Subscriber style. The connection between a publisher and a subscriber is realized by means of *Topics*, which are named buses over which nodes exchange data. They are typed by messages (ROS data structure). A Node publishes on a Topic through a Publisher while subscribes to a Topic through a Subscriber. Topics allow several publishers and several subscribers.

Service Server and Service Client are interfaces (respectively provided and required) that can be used for implementing a communication based on the Client-Server style. The connection between a Service Server and a Service Client is realized by means of a *Service*, which defines the contract between

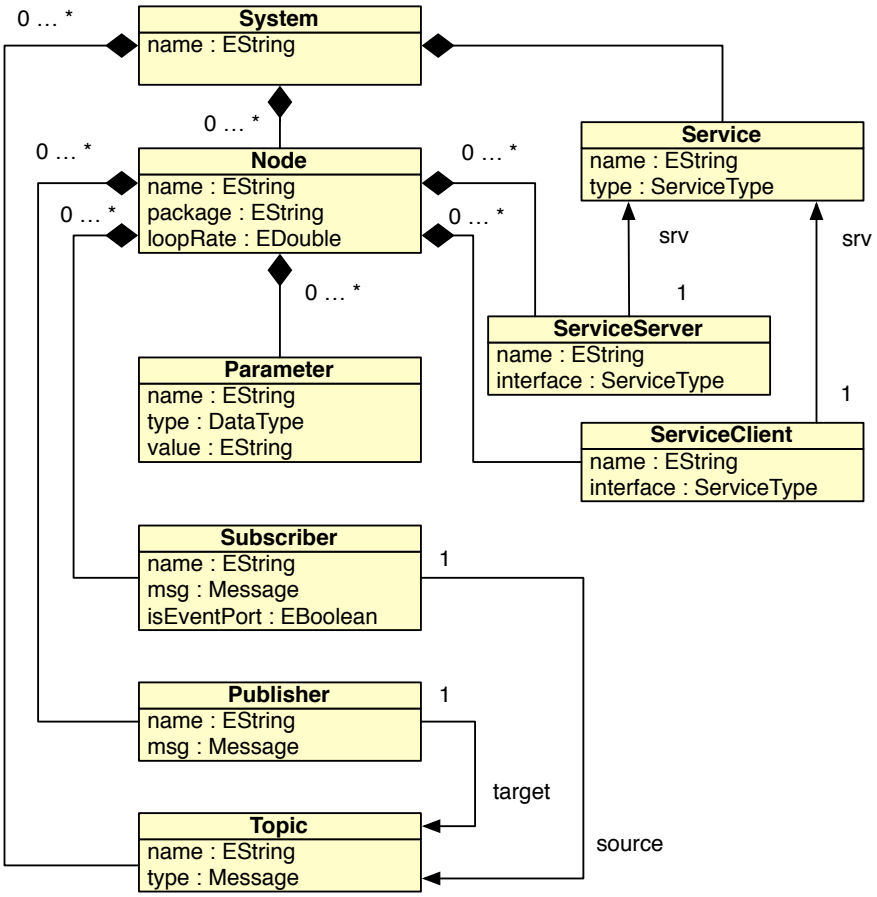

Figure 4.4: The ROS component meta-model

the two entities in terms of two messages: the request and the response.

Finally Nodes can be configured through their *Parameters*, which are the ROS equivalent of the Properties.

It has to be noted that the *Connection* defined in the Abstract meta-model can be modeled in ROS by means of the triplet *Publisher.target*, *Topic*, *Subscriber.source* or the triplet *ServiceClient.service*, *Service*, *ServiceServer.service*.

4.2.3 The Orocos component meta-model

The Component meta-model of Orocos is depicted in figure 4.5[2].

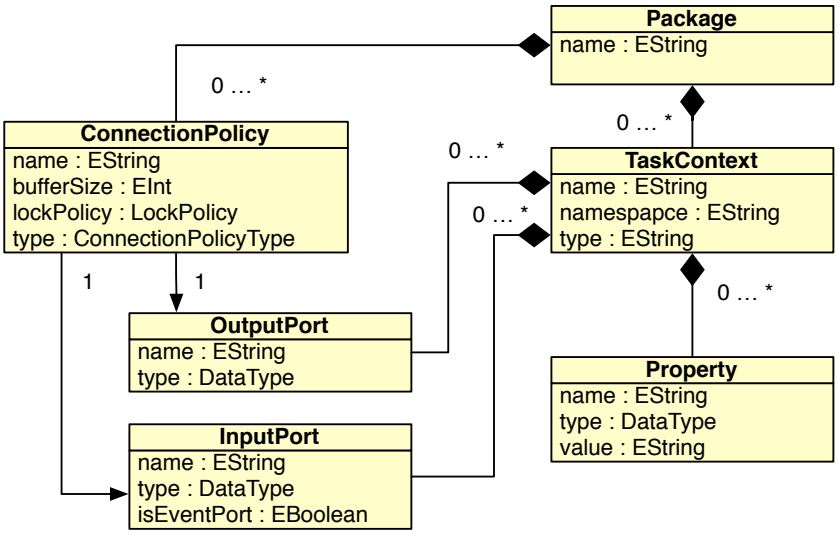

Figure 4.5: The Orocos component meta-model

An Orocos *Package* is the equivalent of a *System*. It contains *Task Contexts* and *Connection Policies*.

Orocos components are implemented as extension (inheritance) of the base class *TaskContext*, have *Properties* and their own thread of execution, and can be deployed as objects that share the same address space or as executables that communicate using the CORBA middleware. Different implementations of the same component are identified by means of two attributes: the *namespace* and the *type* (class name).

Orocos components have *Input Ports* and *Output Ports*, which correspond to required and provided interfaces respectively. They are used for data-flow communication between connected components.

Components with real-time, deterministic and cyclic behavior get fixed and cyclic time budgets for computation. Within a computation cycle they read data from the *Input Ports*, must reach stable intermediate states, and

[2]The Orocos meta-model is a revised version of the Ecore model available at `http://www.best-of-robotics.org/bride/rtt.html`

write results on the *Output Ports*.

Components with reactive behavior define *Event Input Port*, which trigger the component computation upon arrival of new data.

A Connection Policy is the Orocos equivalent of a Connection. It allows the developer to defines the policy of the data-flow communication by means of two parameters: *type* and *lockPolicy*. The first one specifies whether the data sent on a connection are stored in a buffer or not (i.e. each new data replaces the old one). The second parameter defines the lock policy, which can be *lock free*, *locked* or *unsync*.

4.2.4 The SCA component meta-model

A simplified version of the SCA Component meta-model is depicted in figure 4.6[3]. This simplified model reports only the classes that are used in the tools presented in this chapter, however the tools leverage on the original SCA meta-model.

A SCA *Composite* is the equivalent of a *System* and is a graph of *Components* and *Wires*.

A Component interface is defined in terms of *Component Services* and *Component References* (aka Services and References), which are used for implementing a communication based on the Client-Server style. Services and References are typed by means of an *Interface* (e.g. a Java Interface), which defines the signature of the methods provided or required respectively by the Services and the References. Services and References can also be associated with one or more Bindings, which allow different remote software to communicate with them in different ways, i.e. the WSDL binding to consume/expose web services, the JMS binding to receive/send Java Message Service, the Java RMI binding for classical caller/provider interactions of remote components.

A Component implementation is defined by means of a subclass of the abstract class *Implementation*. An example of implementation available in

[3]The types *NCName*, *AnyUri* correspond to *java.lang.String* while *QName* to *java.xml.namespace.QName*

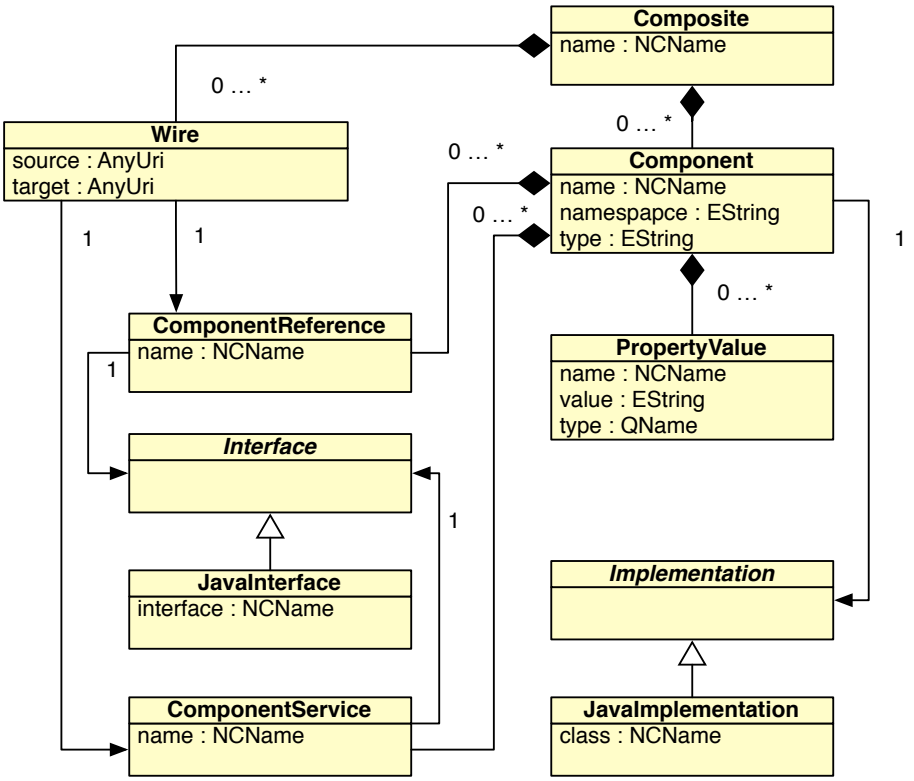

Figure 4.6: The SCA component meta-model

SCA is the *JavaImplementation*, which is characterized by the class name (including the package). In order to be valid the implementation has to implement the interfaces associated to the component Services. Finally a Component can be configured by means of its *Property Values*, which are the equivalent of the properties.

A Wire is the SCA equivalent of a Connection, whose parameters *source* and *target* correspond to the concatenation of component name and service or reference name (e.g. *component.name/service.name*). Wires can be created only between Services and References that refer to the same interface.

4.3 Variability modeling

The feature meta-model depicted in figure 4.7 is designed according to the feature models specification introduced in section 3.4 and defines the following elements.

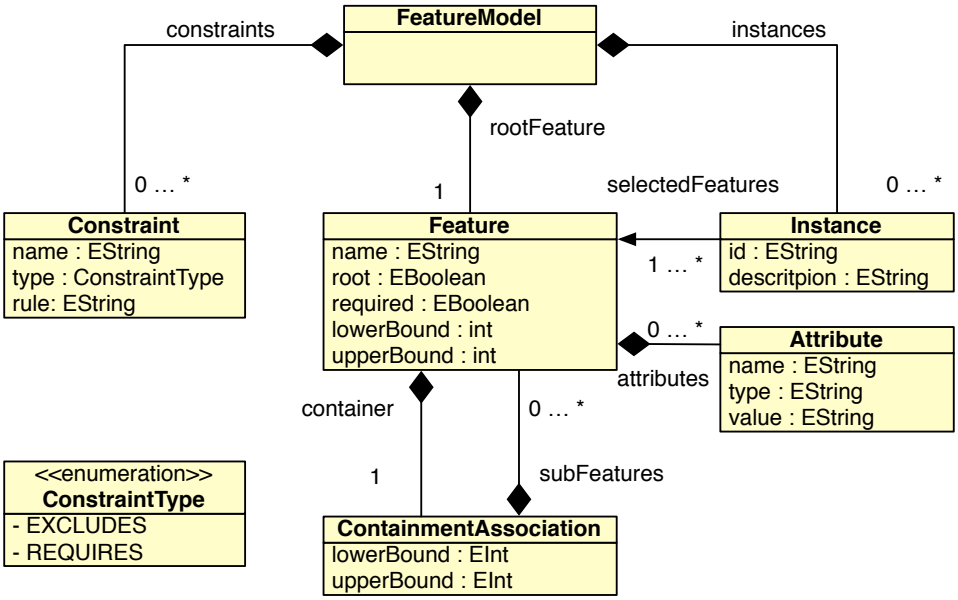

Figure 4.7: The feature meta-model

The root entity *FeatureModel* is a class that encapsulates a tree of *Features* and a set of *Constraints*. It also encapsulates a set of *Instances* that can be saved and reused with the feature model.

- *Feature*. Features are defined by a name and by the boolean attribute *required*, which is true if the feature is mandatory. Features also have another boolean attribute (*root*, true only for the root feature) and two integer attributes (*lowerBound* and *upperBound*), which represent the feature cardinality [38]. Finally, features contain *Attributes* and *ContainmentAssociations*.

- *Attribute*. Attributes can be used for representing information that are important but not so relevant to be represented as features. They have been introduced by [38].

- *ContainmentAssociation.* The containment associations are used for representing the containments of children features. They are defined by two integers (*lowerBound* and *upperBound*) that allow the representation of the standard containments (alternative and or) and of the cardinality containments [34]. The containment associations contain the features that are part of the containments.

- *Constraint.* Constraints are defined by a name and a rule. The rule is a string of the form *[Feature Type Feature]*, where Type can assume the values *requires* or *excludes*.

- *Instance.* Instances represent a specific configuration of the model and are defined by a name and a description. An instance contains references to the set of selected features.

The meta-model described above is modeled in Ecore and is part of the Eclipse Feature-Model Plugin described below, which provides a graphical editor for the design of the Feature Models. The plugin is open source and can be downloaded and installed from Github[4].

4.3.1 The feature model editor

The feature model editor allows the creation of feature models that conforms to the meta-model presented above. It is a graphical editor realized by means of the Eclipse GMF tools. Figure 4.8 depicts the editor and shows how the feature models are represented. The figure reports a feature model that describes the possible configurations of a robust navigation product line.

The graphical representation is conforming to the standard convention of the feature diagrams, except for the group containments. In particular the different entities defined in the meta-model are represented in the following way.

- Features are represented by means of a white box that contains the name of the feature. The blue box represents the concept (root feature).

[4]http://robotics-unibg.github.com/VARP

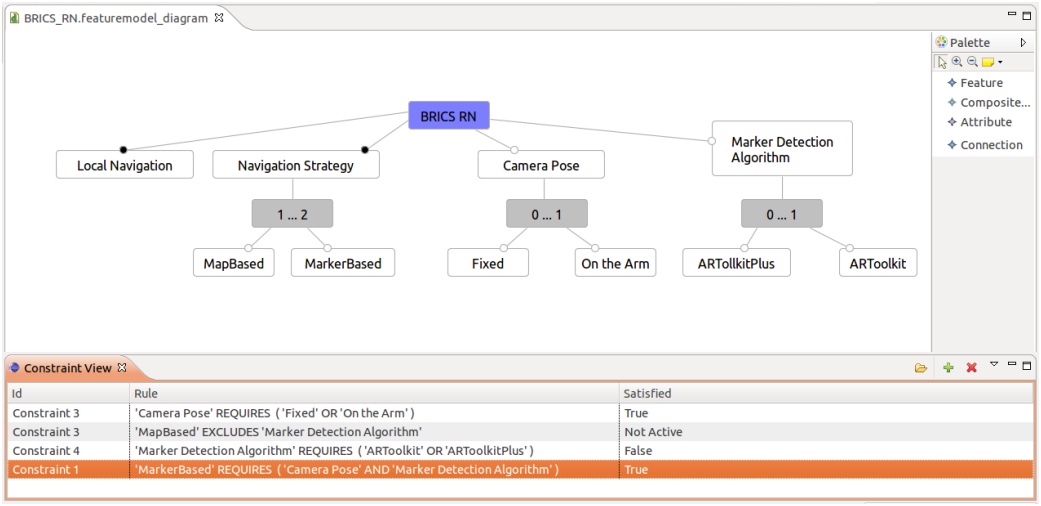

Figure 4.8: The Feature Model Editor

- Mandatory features are represented by means of a black circle on the top whereas optional features by means of a white circle.
- Containment Associations are represented by means of a grey box that shows the lower and the upper bound of the cardinality.
- Attributes are represented by means of a cyan box that contains the name of the attribute.

The editor also provides the possibility of defining constraints by means of the dialog window depicted in figure 4.9. It forces the user to create rules that are syntactically correct, for instance it doesn't allow him to insert two logical operators without a feature between them. The constraint presented in the example means that if the *MarkerBased* feature is selected then also a sub-feature of *Camera Pose* and a sub-feature of *Marker Detection Algorithm* have to be selected.

4.4 Variability resolution

The *Template System Model* and the *Feature Model* represent orthogonal concerns, i.e. the structure of the component-based system and the hierarchical

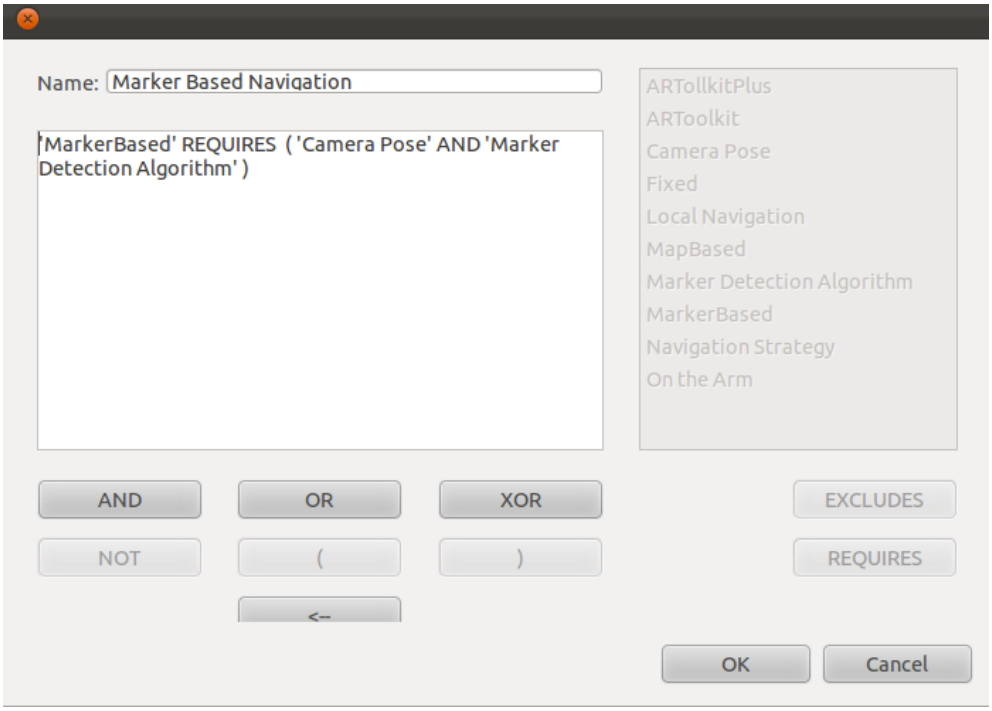

Figure 4.9: The Constraints Editor

organization of functional and non-functional aspects. For this reason, these models do not have dependencies to each other, since they should evolve independently.

The *Resolution Model* represents the bridge between the *Template System Model* and the *Feature Model*. It defines a set of *Resolution Elements*, which specifies the relationship between a feature defined in the *Feature Model* and a set of architectural elements defined in the *Template System Model*.

Each *Resolution Element* contains one or more *Required Elements* and *Transformations* (see subsection 3.6) and refers to a feature. The resolution element is applied on the *Template System Model* only when the corresponding feature is selected to be part of the instance that is created by the System Integrator and is sent to the resolution engine.

The meta-model defines two types of required elements, *RequiredComponent* and *RequiredConnection*, and three types of transformations.

- ***Implementation Transformation***: it specifies a link to a given

component in the Template System Model and the implementation that has to be associated to it.

- ***Property Transformation***: it specifies a link to a property of a given component and the value that has to be assigned to it.
- ***Connection Transformation***: it specifies a set of new connections that have to be created between pairs of components.

As describe in section 3.3 for the Component meta-model, also in the case of the resolution meta-model two levels of abstraction have been defined: the first defines software framework independent concepts (see the resolution Model depicted in figure 4.10) while the second specifies three software framework specific resolution meta-models.

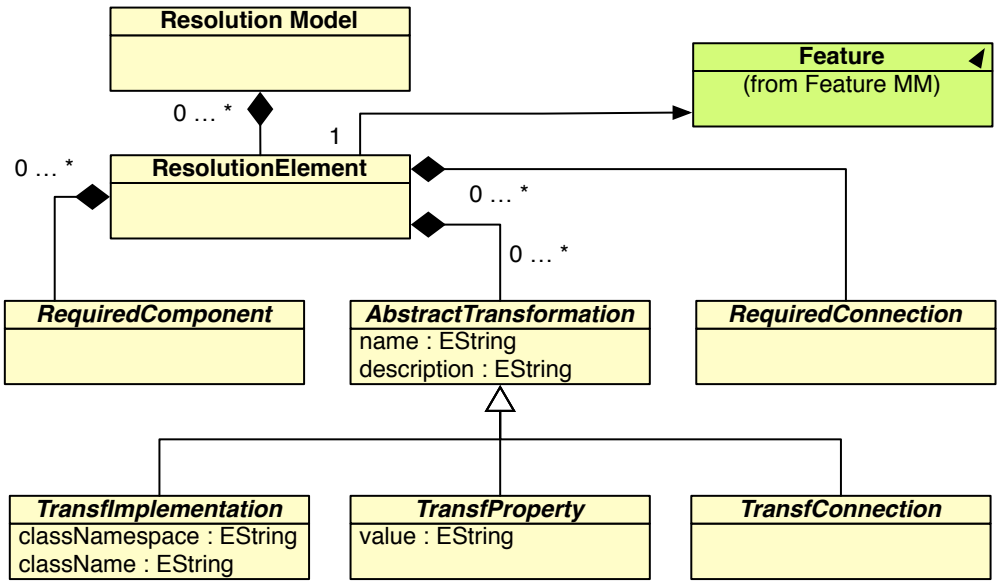

Figure 4.10: The resolution meta-model

4.4.1 The ROS resolution meta-model

Figure 4.11 illustrates how the resolution meta-model presented in this section has been specialized for ROS. Yellow classes, whose name starts with the

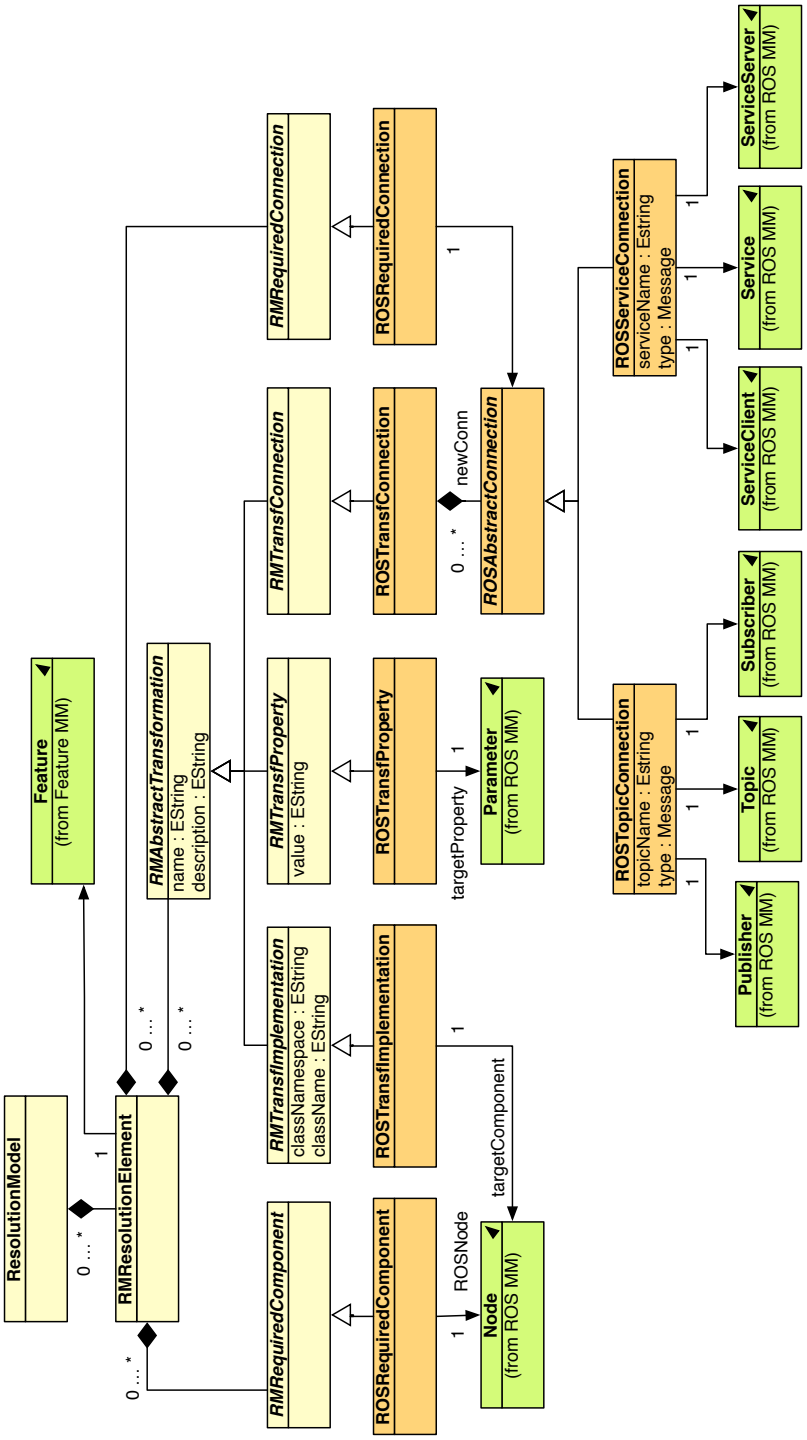

Figure 4.11: The ROS resolution meta-model

letters *RM*, are defined in the resolution meta-model (including the class *ResolutionModel*). Green classes with the arrow on the top right corner are entities defined in the ROS Component meta-model (see figure 4.4) and in the Feature meta-model (see 4.7). Finally orange classes, whose name starts with the word *ROS*, are part of the *ROS resolution meta-model*.

The classes *ROSRequiredComponent* and *ROSRequiredConnection* are used to define ROS Nodes and Connections that are required in the *Configured System Model* when the feature associated to the Resolution Element is selected. ROS connections can be defined by means of the triplet *Publisher.target*, *Topic*, *Subscriber.source* or the triplet *ServiceClient.service*, *Service*, *ServiceServer.service*. The class *ROSTopicConnection* models the first triplet while the class *ROSServiceConenction* the second. They both extend the abstract class *ROSConnection*.

The class *ROSTransfImplementation* specializes the abstract class *RMTransfImplementation* and has a pointer to the Node for which it is necessary to set the implementation. The attribute *classNamespace* specifies the package while *className* the node name.

The class *ROSTransfProperty* specializes the abstract class *RMTransfProperty* and provides a pointer to the Parameter that has to be set.

Finally the class *ROSTransfConnection* specializes the abstract class *RMTransfConnection*. The class provides information about the Connections that have to be created. This information is provided as a collection of *ROSConnections*.

4.4.2 The Orocos resolution meta-model

Figure 4.12 illustrates how the resolution meta-model has been specialized for Orocos. Yellow classes, whose name starts with the letters *RM*, are defined in the resolution meta-model (including the class *ResolutionModel*).Green classes with the arrow on the top right corner are entities defined in the Orocos Component meta-model (see figure 4.5) and in the Feature meta-model (see 4.7). Finally orange classes, whose name starts with the word *RTT*, are part of the Orocos resolution meta-model.

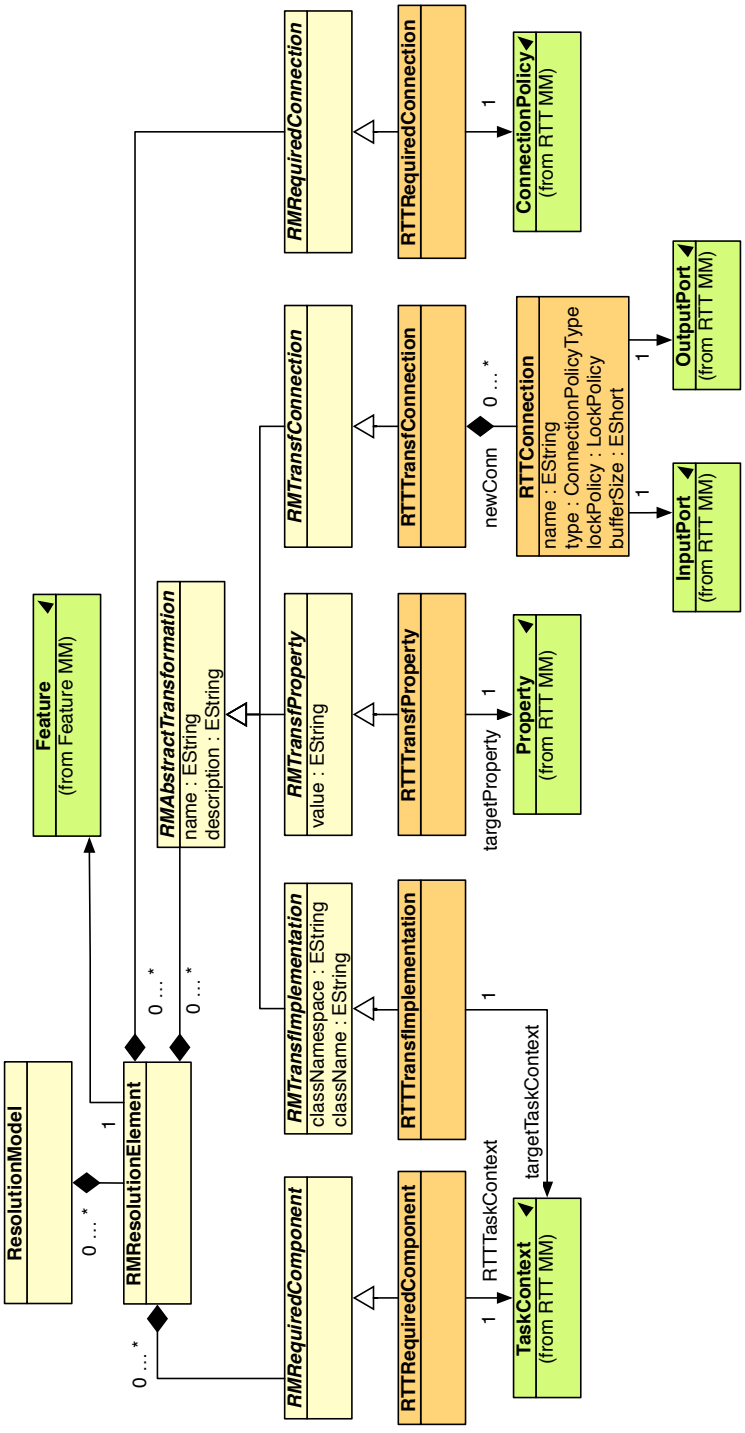

Figure 4.12: The Orocos resolution meta-model

The classes *RTTRequiredComponent* and *RTTRequiredConnection* are used to define Orocos Task Contexts and Connection Policies that are required in the in the *Configured System Model* when the feature associated to the Resolution Element is selected.

The class *RTTTransfImplementation* specializes the abstract class *RMTransfImplementation* and has a pointer to the Task Context for which it is necessary to set the implementation. The *classNamespace* and *className* attributes of the implementation are defined in the parent class.

The class *RTTTransfProperty* specializes the abstract class *RMTransfProperty* and provides a pointer to the Property that has to be set.

Finally the class *RTTTransfConnection* specializes the abstract class *RMTransfConnection*. The class provides information about the Connections that have to be created. This information is provided as a collection of *RTTConnections*, which specify their type (data or buffer), the Input Port, the Output Port, the lock policy (lock free, locked or unsync) and possibly the buffer size.

4.4.3 The SCA resolution meta-model

Figure 4.13 illustrates how the resolution meta-model has been specialized for SCA. Yellow classes, whose name starts with the letters *RM*, are defined in the resolution meta-model (including the class *ResolutionModel*).Green classes with the arrow on the top right corner are entities defined in the SCA Component meta-model (see figure 4.6) and in the Feature meta-model (see 4.7). Finally orange classes, whose name starts with the word *SCA*, are part of the SCA resolution meta-model.

The classes *SCARequiredComponent* and *SCARequiredConnection* are used to define SCA Components and Wires that are required in the *Configured System Model* when the feature associated to the Resolution Element is selected.

The class *SCATransfImplementation* specializes the abstract class *RMTransfImplementation* and has a pointer to the Component for which it is necessary to set the implementation. In this case the attribute *classNamespace*

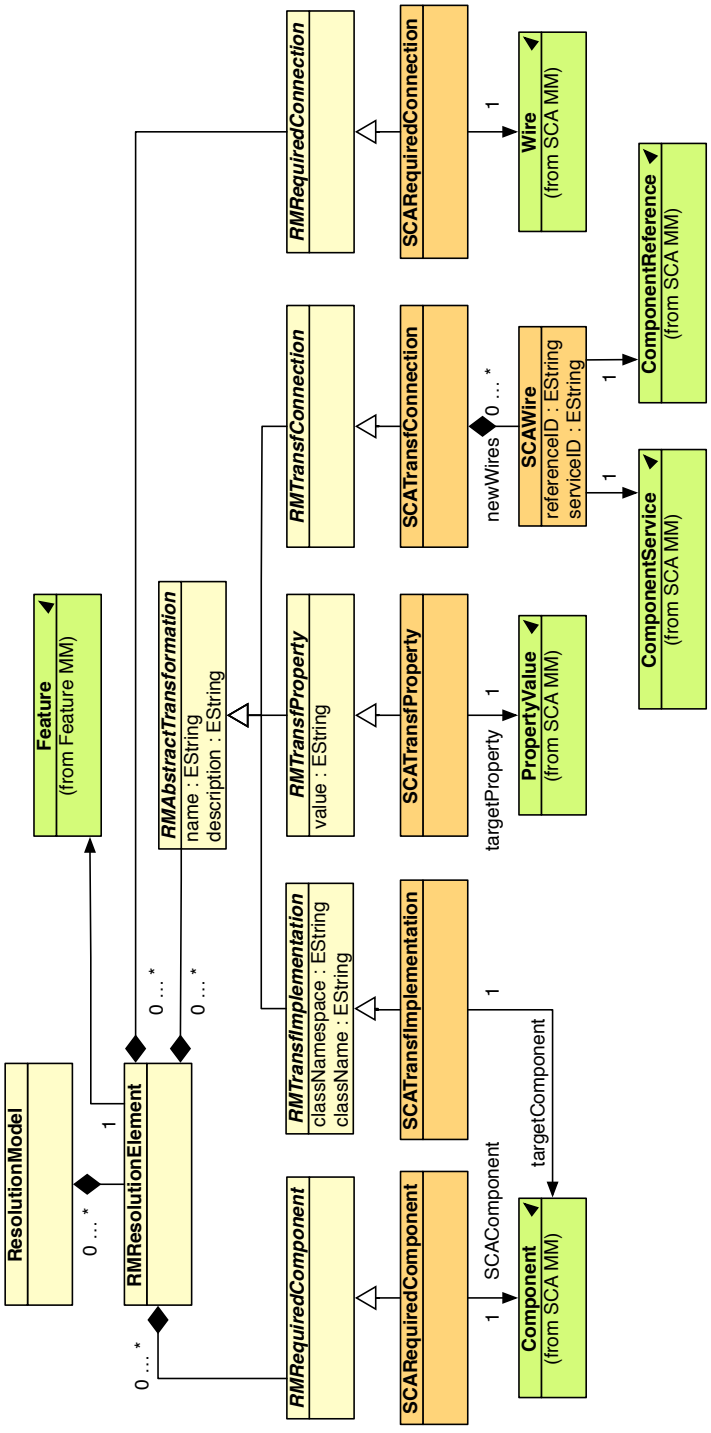

Figure 4.13: The SCA resolution meta-model

defines the package name while *className* the name of the class.

The class *SCATransfProperty* specializes the abstract class *RMTransf-Property* and provides a pointer to the PropertyValue that has to be set.

Finally the class *SCATransfConnection* specializes the abstract class *RMTransfConnection*. The class provides information about the Connections that have to be created. This information is provided as a collection of *SCAWire*s, which specify the reference and the services that have to be connected.

4.5 Product Derivation

In the last stage of the development process the *Feature Selector* editor allows the System Integrator to create instances of a feature model and to generate the corresponding *Configured System Model*. Creating an instance consist of selecting the sub-set of features, from all the features that are defined in the model, which satisfy the desired functionality for the application the has to be deployed. This selection has to contain all the mandatory features, satisfy the cardinality of the containments and respect all the constraints defined on the model. Figure 4.14 depicts the Feature Selector. It is composed of two parts: the visual representation of the model and the instance view.

The instance view shows the information about the existent instances and offers a set of commands that allow the System Integrator to create, remove and select instances. The third column provides information about the instance that is currently showed in the visual representation.

Selected features are drawn in green. The System Integrator can select a feature for being part of the instance by simply selecting it and using a button in the toolbar or an entry of the *Feature Models* menu.

Once the System Integrator has completed the selection of the features, the instance can be validated and the *Configured System Model* generated by means of the resolution engine, which is described below.

The instance validation involves the verification of the fulfillment of the following requirements:

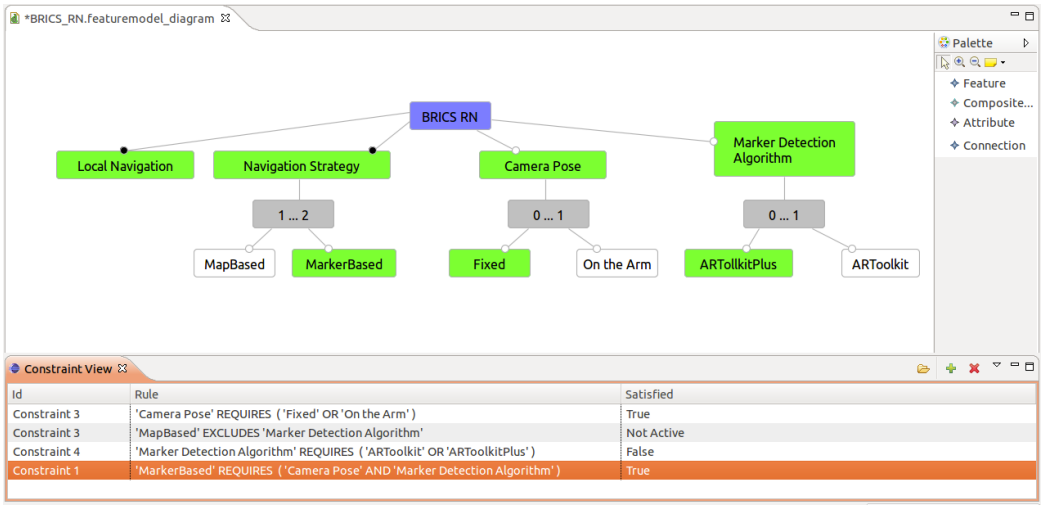

Figure 4.14: The Feature Selector

1. All the mandatory features have to be selected to be part of the instance.

2. All the cardinalities of the containments have to be satisfied, which means that the number of the selected sub-features of a containment has to be greater than or equal to the lower bound and less than or equal to the upper bound.

3. All the explicit constraints have to be respected. The constraint checking is done by using an external open source library called MVEL [39], which resolves the two members of each constraint rule. After that, if the constraint type is "require", the constraint checker controls that both the members are true. In case of an "exclude" constraint instead when the first member is true the constraint checker controls that the second is false.

4.5.1 The resolution engine

As already explained in section 4.1 the resolution engine receives as input a set of Template System Models, a set of Resolution Models and an Instance of the Feature Model. In order to produce as output the Configured System

Model it performs the following actions:

1. It controls if a feature called *Component Model* is defined. This is a key-feature that describes which Resolution Model has to be used for deploying the System Configuration. No Required Elements should be associated to this Feature because the resolution engine uses it only for understanding which Resolution Model has to be used. In the case that this feature is not present the process terminates with an error. On the contrary the resolution engine controls that the Template System Model and the Resolution Model for the selected software framework have been provided.

2. It creates a copy of the Template System Model. This copy is the first versions of the Configured System Model and will be modified during this process. An empty list of required components and an empty list of required connections are instantiated.

3. It iterates all the Resolution Elements defined in the Resolution Model and for each of them controls if the associated feature has been selected for the Feature Model Instance. If the feature is not selected, then the Resolution Element is discarded. In the opposite case the transformations associated to the Resolution Element are executed and the required components and connections are inserted respectively in the required component list and the required connection list. The connections created by the Connection Transformations are inserted as well in the list of the required connections. In the same way the components pointed by the Implementation Transformations and the components containing the properties pointed by the Property Transformations are inserted into the required component list.

4. It removes from the Configured System Model all the components and the connections that are not contained in the required component and connection lists.

5. The *Configured System Model* customization is finished. The resolution

engine validates the output model in order to ensure that all the rules defined in the corresponding Component meta-model are satisfied.

At the end of this process the *Configured System Model* can be deployed with the corresponding software framework.

It is important to note that the Resolution Model and the Resolution Engine are designed in such a way that the transformations associated to the Resolution Elements can be executed in any possible order, without the risk that two transformations will produce conflicting effects. For example it is not possible that a transformation tries to create a connection that involves the port of a component that is not yet defined in the Configured System Model or that will be removed during the last step. The same is true for a transformation that tries to set an implementation or the value of a property.

4.6 Related works

This section introduces the related works regarding the Feature Models Plugins and the variability resolution.

4.6.1 Feature models

Despite along the years Feature Models have gained popularity, only few attempts of providing a set of tools, which support the design of these models, can be found in literature. Moreover some of those are not open source. This section provides a survey on the projects described in literature. It will focus on the eclipse-based tools, because with respect to the standalone tolls they can be more easily integrated with other tools. The goal of the survey was founding an open source and eclipse-based Feature Model Plugin, providing the following features.

- Conformity to the standard specification of the Feature Models that was defined in literature for what regards the features, the containments and the constraints (see subsections 3.4.1, 3.4.2 and 3.4.3).

- The presence of a graphical tool, based on a formal model, which allows the user to define Feature Models in form of Feature Diagrams in a simple and user-friendly way.
- The presence, in the same graphical editor, of a functionality that allows the user to select a set of features directly from the same feature diagram defined previously and allows the verification of the constraints fulfillment.

This section uses the term graphical editor for defining an editor that allows the representation of the Feature Models in a form similar to the feature diagrams depicted in figure 3.2.

The Eclipse Modeling Framework Technology project (EMFT)[40], which aims to provide a set of new tools that extend or complement the Eclipse Modeling Framework (EMF)[25], provides a Feature Model plugin called EMF Feature Model [41]. This plugin is still in the incubation stage and it is based on two meta-models. The first one describes the rules for modeling the variability and designing Feature Models, whereas the second the rules for creating instances of the Feature Models. The major drawback of this plugin is the dependency to pure::variants [42] for providing a graphical representation and the constraints evaluation. Pure::variants is a complete Eclipse plugin, however its use has been avoided due to the commercial license. Indeed, it is free available only in a limited version.

Another interesting plugin, which is still in development, was designed at the university of Waterloo [43]. In contrast to the proposal of [41], this plugin uses a single meta-model for representing both the Feature Model and its configurations. They provide an XPath support for the definition of the constraints and also a constraints evaluation engine. This plugin is completely open source but it doesn't provide a graphical editor.

The P&P Software GmbH and the ETH-Zurich developed an open source Feature Model plugin called XFeature [44]. This plugin provides a graphical editor, an XPath constraints definition support and a constraints evaluation engine. The editor is in general very complete but the editing of the Feature Model is not really user friendly. For example a tools palette is not provided

and the user has to edit the model by continuously using the context menu. Furthermore the plugin allows the users to define an instance only by creating a new feature diagram in which only the features defined in the Feature Model can be inserted.

Finally one of the most complete and open source plugin found in literature is FeatureIDE [45]. It provides a lot of functionalities, for example a graphical editor which is completely compatible with the standard notation (it was the only one found during the survey), a statistics view that collects information about the number of features, the number of variants and the number of possible configurations and last but not least the possibility of comparing the current version with the last saved version in order to reason about how the changes in the model will affect the product line. Despite this editor is very complete, it has some drawbacks.

- Like the previous plugin it doesn't provide a tools palette for editing the model.

- It provides a tree representation for the selection of the features that are part of an instance. However it doesn't allow the users to do this operation by using the standard feature diagram representation, which is more intuitive.

- It doesn't explicit allow the definition of the exclude constraint (the workaround is defining "*A excludes B*" in the form "*not (A and B)*").

This survey highlighted that no one of the plugins fulfills the three requirements defined above. In particular, for what regards the third point, most of the available plugins allow the selection of a configuration only by using a tree-view of the model or by recreating a new feature diagram in which the user has to insert some of the features defined in the Feature Model. In order to overcome the limitations of the available tools, the meta-model and the plugin described in this chapter have been developed.

4.6.2 Variability resolution

The Common Variability Language (CVL)[46] is "a domain independent language for specifying and resolving variability", which has been proposed as OMG standard. The CVL approach is based on three models: (a) the *Base Model*, which describes a software product line (SPL) in terms of architectural elements and can be written in any MOF compliant language; (b) the *Variability Model*, which models the variability of the SPL and defines how the Base Model has to be modified according to the selected variants (it is written in CVL); (c) the *Resolution Model*, which is an assignment of values to the variants defined in the Variability Model.

The Variability Model is a tree made of *VSpec* elements. Four kinds of VSpecs are defined: *Choice*, *Variable*, *VClassifier* and *Composite VSpec*. *Choice* is the analog of feature and during the resolution can assume two values: selected or not selected. *Variable* is a VSpec whose resolution requires a value for its specific type (e.g. int). *VClassifier* is a VSpec that can be instantiated more times. Finally *Composite VSpec* allows the hierarchical composition of VSpec. A Variability Model made only of Choices is semantically equivalent to a Feature Model.

The Variability Model associates *Variation points* to VSpecs. A Variation Point defines how the Base Model has to be modified according to the value that is assigned to the corresponding VSpec during the variability resolution. Several variation points are available, which allow any possible modification on the Base Model. In particular the derivation of Composite VSpec allows the resolution of several variation points by assigning a value to only one VSpec (i.e. it is more or less the equivalent of what it is possible to do with the approach presented in this thesis by assigning more than one transformation to a single Resolution Element).

CVL differs from the approach described in this thesis in several points. (a) They defined a new language (CVL) for modeling the variability while the approach described in chapter 3 uses the well-known Feature Diagram representation. (b) While CVL uses a unique model for modeling variability and transformations in this thesis the information is separated in two different

models (the Feature Model and the Resolution Model). In this way the same Feature Model can be used for specifying different Resolution Models and the Feature Model Plugin can be used independently from the transformations (e.g. for documenting the variability of a system). (c) Unlike CVL, which has the goal to be as general as possible, the approach here presented is expressly designed for component-based software frameworks that offer the variability mechanisms described in section 4.2. For this reason the transformations that can be applied on the Template System Model leverage on those mechanisms and only allow the modification of specific architectural elements (component implementations, properties and connections). (d) The Template System Model contains all the possible components that can be used in all the possible system configurations. For this reason components are never added during the variability resolution. This characteristic of the System Template Model allows us to execute transformations in any possible order. The Feature Models constraints (both explicit and containment constraints) and the Required Elements ensure that the transformation will not produce conflicting effects. (e) The Variability Model here proposed is completely orthogonal to the Template System Model and allows the association of highly complex transformations to a single feature. A feature doesn't have a one to one relationship with specific element of the Template System Model. Differently, in all the examples reported in [46] there is an implicit one-to-one relationship between VSpecs and architectural elements defined in the Base Model.

Lee at al. [47] propose an approach based on a feature oriented product line engineering for the development of reusable service-based systems. In the first step of their approach the feature oriented domain analysis is used for identifying candidate reusable services, dependency among these services and variation points for runtime binding. In a second phase, called service analysis, the identified services are then distinguished in two categories: molecular and orchestrating services. The first group provides computational services that are reusable among several products. The second group instead provides behavioral services, which are typically product-specific.

The proposed approach differs from the one proposed in this thesis in two

points. First of all [47] uses Feature Models for the domain analysis while the approach here described uses Feature Models also for the automatic system configuration deployment. Furthermore, [47] applies only to service oriented architectures, while the approach documented in this thesis can be adopted for data-flow (e.g. Orocos) and publish-subscriber (e.g. ROS) styles.

Cirilo at al. have defined a model-based approach for variability resolution similar to the one proposed in this thesis. In [48] they apply this approach to multi-agent systems while in [49] the same approach is applied to OSGi and Spring component based systems. The approach is based on three models: (a) an Architecture Model, which defines a visual representation of the implementation elements (classes, aspects, bundles, beans); (b) a Feature Model, which describes the variability of the system architecture; (c) a Configuration Model, which defines the mapping between the features and the implementation elements.

The approach proposed in this thesis differs from the Cirillo's approach in several points. (a) The architectural elements defined in [48] vary from classes to code fragments while in this thesis the approach models more abstract elements such as components, interfaces and connections. (b) In [49] also OSGi and Spring components are modeled as architectural elements. However, despite bundles and beans define dependencies, the concept of connection is not explicitly defined like in the Abstract Component Model here described, but it is only implicitly implemented in the code. For this reason it is not possible to change the connections between components in the same way it is described in chapter 3. (c) As well as [47], Cirillo's approach applies only to service oriented architecture and is not usable with the other architecture styles, which are widely spread in the Robotics domain.

5 Refactoring

Today, a huge corpus of software applications, which implement the entire spectrum of robot functionality, algorithms, and control paradigms, is available in robotic research laboratories and potentially could be reused in many different applications. For various reasons (e.g. they defines different interfaces, they use different data structures), the interoperability between different implementations or their extensions towards novel applications is limited or would require high efforts.

This chapter introduces the concept of refactoring and presents a set of guidelines that describe how to refactor existing software libraries in order to facilitate their encapsulation in reusable components, which can be then used for building Robotics Product Lines as described in the chapter 3.

Moreover this chapter illustrates, by means of a case study, how the theoretical guidelines have been applied for analyzing open source implementations of best practice libraries for motion planning and refactoring one of them (CoPP [50]).

5.1 Code Refactoring

As introduced in section 3.2, the refactoring is the process of changing a software system in order to improve its internal structure without changing its external behavior [28].

The refactoring process implies removing of duplicate code, simplification of too complex logic flows and improving of code readability. Typically the process evolves by following little and simple steps during which the code is modified. In order to reduce the probability of introducing new bugs after each step the refactored code is tested.

In the book [51], Joshua Kerievsky explained by means of four motivations why the refactoring process could be useful.

1. *Make it easier to add new code.* When we need to add a new functionality to our software we can operate in two different ways. The first consists in doing it without considering how this new feature fits with the existing design. The second possibility instead consists in modifying the existing design in order to facilitate the integration of new functionalities. The first choice is good when we have little time. Instead, if we have enough time and (more important) if we think that we will have to introduce new features also in the future, then the second choice is definitely better.

2. *Improve the design of existing code.* Iterative executions of refactoring process make easier to remove what Fowler calls "Bad smells" and then to improve the code design. Bad smells are certain structures in the code that suggest the possibility of refactoring. In their books Fowler and Kerievsky describe and explain how to remove these smells.

3. *Gain a better understanding of code.* Sometimes when we analyze code can be very hard to understand how it works. In this case add comments may not be enough for those who will have to modify the code in future. They may still not understand it. The best solution is rewriting the code and commenting it. For example a good practice consists in choosing explicative names for variables and methods. Ideally the name of a method should explain which functionality it realizes. Another good solution instead consists in dividing a long method into shorter methods. About the readability of code Fowler said: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand".

4. *Make coding less annoying.* Refactoring not only makes code more readable but also can make it less annoying to work with. In this way it is simpler and quickly modify the code. For example, when a class has too many responsibilities, every time we need to modify the code or integrate new functionalities, we have to deal with this complex class. As result we spend more time than necessary in order to complete our work. In this case it is convenient rewriting the code and dividing the complex class into simpler classes. This operation will allow a significant saving in terms of time during future works on the code.

5.1.1 The refactoring process

The goal of the refactoring stage, which was described in section 3.2, is transforming different implementations of the same algorithm specification in a software library that is Software Framework independent and provides harmonized interfaces and data structures. The refactoring process, which has to be applied in order to achieve this goal, is made of three steps.

1. *Domain analysis.* Domain analysis concerns the task of getting inside into the problem domain. This step consists in an exhaustive research of the best practice algorithm implementations, which were developed in the interested domain. These algorithms are then deeply analyzed in order to identify their commonalities and their differences.

2. *Harmonization.* Harmonization concerns the definition of the architecture of a software library, which allows the integration of the implementations considered in the previous point. This step defines: (a) the harmonized interfaces, (b) the stable data structures and operations, (c) the variation points. Typically this architecture can be represented through a UML class diagram, where stable data structures are identified by means of concrete classes and variation points by means of abstract classes. Variation points offer basic operations (e.g. path planning functionality) that are implemented in a specific way by different variants (e.g. probabilistic roadmap or rapidly-exploring random trees).

Stable data structures instead implement concepts that are hardly going to change over the time. During this stage the designer typically apply a set of refactoring patterns, which are described in section 5.2.

3. *Integration.* Integration concerns the development of the architecture designed in the previous step. During this step the code of the considered implementations is partially rewritten in order to fit with the common data structures and operations. Each different implementation of an algorithm is encapsulated in a specific variant that implement a variation point. In order to check that the external behavior is unchanged the code is then tested.

Figure 5.1 depicts the refactoring process that transforms a set of algorithms implementations adhering to the same specification into a harmonized software library.

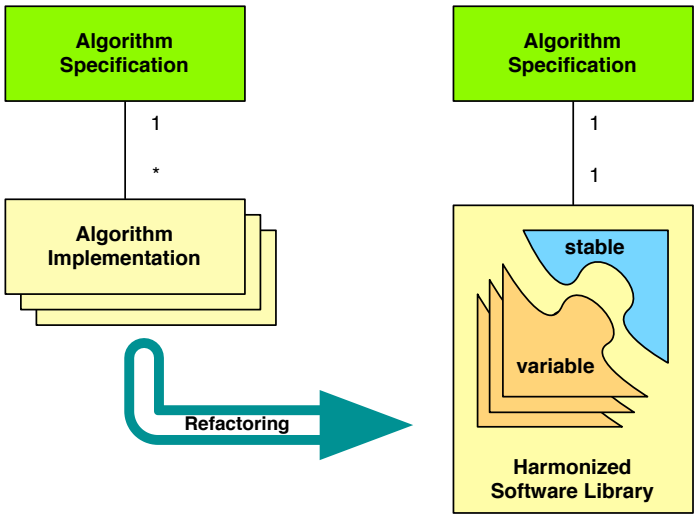

Figure 5.1: The Refactoring process

The process can be illustrated through the following simple example, which will be deeply analyzed in the case study sections. At the state of the art many path planning algorithms are available, for example the probabilistic roadmaps and the rapidly-exploring random trees techniques. By analyzing

their implementations some commonalities can be identified: they define the concept of path and a method (called *pathPlanning*) that computes an optimal path from an initial configuration to a final one. The functionality provided by this method is defined in the specification of each path planning class hierarchy.

In this case the domain analysis (first step of the refactoring process) identifies the commonality between the two algorithms, which is the path concept, and the differences, that are the two implementations of the *pathPlanning* method.

The second step (harmonization) defines the software architecture. In this case the stable part is represented by the path (data structure) and the variation point by the interface of the *pathPlanning* method. The variants are instead represented by the different implementations of this method.

Finally the integration stage releases the working version of the harmonized software library.

A developer may use it by choosing one of the available variants or by adding a new one. In fact, in order to develop a new algorithm the software library can be easily extended by implementing a new variant, or rather a new implementation of the *pathPlanning* interface. Hence the developers may reuse the stable parts of the software library and focus their effort on the development of the new variants.

5.2 Refactoring patterns

Refactoring patterns have been documented during the last years in several Software Engineering books, which describe them and explain how they can be applied for improving the design of existing code (see for example [51, 28, 52]). This section briefly introduce some of the patterns that have been applied during the refactoring of the motion planning and robust navigation libraries (see respectively case studies in section 5.3 and section 6.3), leaving to the reader the possibility of deepening and applying them.

5.2.1 Responsibilities redistribution

The refactoring patterns *Move Behavior Close to Data*, *Eliminate Navigation Code* and *Split up God Classes* [52, Chapter 9] aim to redistribute the responsibilities among classes and in particular to eliminate pure Data Containers and God Classes.

Data Containers are classes that are simple data structures and have almost no identifiable responsibilities. Use to many data containers makes complex the code navigation and typically results in clients that have to navigate chain of intermediates for reaching the indirect provider (i.e. the intermediate(s) and the indirect provider are unnecessarily coupled).

God Classes are on the contrary classes that assume too many responsibilities. Typically they are never instantiated, contain only static methods and their data structure and behavior have class scope. They represent a complete application. For this reason they are hard to understand and maintain.

Sometimes God Classes and Data Containers happen together, where the God Class is the core of the application and uses Data Containers as pure data structures.

Move Behavior Close to Data

Move Behavior Close to Data aims to resolve the problem of the pure data containers. It suggests to extend the data containers, which defines data structures, by assigning them the responsibilities of creating, initializing, updating, transforming and elaborating encapsulated data (responsibilities that were previously assigned to the intermediates and the indirect client).

Thanks to this pattern the data containers become service providers with well defined responsibilities, avoiding different clients to implement the same operations (i.e. reduction of the duplicated code).

Split up God Class

Split up God Class aims to remove god classes from a software library. All the responsibilities concentrated in a god class are redistributed to the classes that collaborate with it or to new classes appositely defined. Typically some

of the classes that receive new responsibilities are pure data containers. For this reason this pattern can be applied together with *Move Behavior Close to Data*, in order to transform data container in concrete objects.

Thank to this pattern a procedural design can be transformed in an object oriented design and the monolithic class implementing the entire application can be split in a set of classes that provide well-defined responsibilities. As a consequence the ex god class is more stable and easier to understand and maintain.

Eliminate Navigation Code

Eliminate Navigation Code [52, Chapter 10] aims to reduce the impact of changes in the code of the service provider by shifting the responsibilities down a chain of connected classes.

This pattern is applied iteratively along the chain of indirect clients. Starting from the last indirect client, at each step the methods that operate on the data are moved to the container that provides the data itself.

Thanks to this pattern the chains of dependencies between classes can be eliminated. As a result the impact of changes operated on the classes at the lowest level are reduced.

5.2.2 Transform Conditionals into Registration

Another bad smell in the code, which increases the coupling level between the provider of a service and its clients, is the presence of long case statements, which make the code much more fragile. These case statements are typically used for performing a service in a different way according to the attributes of an object that is tested during the condition checking. The case statement is implemented in the client code, which delegates the service execution to a different service provider according to the result of the condition checking.

The pattern *Transform Conditionals into Registration* addresses these situations in which the client is responsible for starting up an external service. The pattern suggests introducing a registration mechanism to which each service provider is responsible for registering itself. The provider clients are

then transformed to query the registration repository instead of performing conditionals.

Thanks to this pattern the client become more flexible and it is not anymore hard-coupled to the existing service providers, which can be added and removed dynamically without modifying the client code.

5.2.3 Family of algorithms

When a set of algorithm implementations conform to the same specification and differs in the behavior (as described in the example in the subsection 5.1.1), the pattern *Strategy* [53] can be used for reducing the coupling between the client of the algorithm and its specific implementation. Strategy defines a family of algorithms, encapsulates each one and makes them interchangeable.

This pattern defines an abstract class called *Strategy* (variation point), which implements the concepts that are common to all the algorithm implementations and provides a common interface for the algorithm invocation and configuration. The subclasses (variants) of the Strategy class represent instead the different algorithm implementations and extend the common interface for providing specific behaviors.

Thanks to this pattern the algorithm implementations can be modified without having to adapt the client code. Moreover new variants can be defined by reusing the commonalities provided by the Strategy class and implementing only the specific behavior.

5.2.4 Inversion of Control

Another aspect that makes a software library more reusable is provide a framework that defines the skeleton of an application and let the users customize its behavior by means of plugins. In this way the users don't have to define the application core and can focus only on the development of new plugins by specializing well defined interfaces provided with the library. In other word the software library defines a set of well defined variation points and leaves to the user the freedom of developing new variants or using the variants provided with the library.

For example a path planning library can provide a framework that defines a variation point for the collision checking functionality. The users of this library have the possibility of customizing the framework by defining a new collision checker and bounding it to the variation point.

This pattern moves the application control from the code written by the user of the library to the library itself, more specifically to the provided framework. For this reason it is called *Inversion of Control* or *Dependency Injection* [54].

In addition to the definition of the variation points the library should define a mechanism for bounding the variation point to the variants. For this porpoise the pattern defines three possible mechanisms that Fowler calls *Constructor Injection*, *Setter Injection* and *Interface Injection*.

5.2.5 Magic Numbers

A frequent bad smell, which makes the code less readable and maintainable, is the presence of the so called *magic numbers*. They are numbers with a special meaning that are encoded in the source code instead of using specific constants, for example the proportional gain of a controller or the number of the joints of a manipulator. The more the same magic number is spread in the code the more the problem is serious. Change the value of this number indeed means changing its value in every point in which it is used in the code.

Fowler suggests eliminating this numbers by applying the refactoring pattern *Replace Magic Number with Symbolic Constant*, whose title is self-explicative.

Alternatively, in order to define a software library that can be easily encapsulated in software components, it is a good practice to replace the magic number, which allows the configuration of the code behavior (e.g. gain parameters), with non-static variable and define for them a set of getter and setter methods. In this way the library or part of it can be easily wrapped in a component and these new variable mapped to a set of component properties.

5.3 Case study - Domain analysis

This sections presents a case study regarding path planning problems with a high number of degrees of freedom, as typically encountered in mobile manipulation tasks. In this context, several classes of probabilistic, sample-based planners have been developed, most notably variants of Probabilistic Roadmaps (PRM) and Rapidly Exploring Random Trees (RRT), amongst others (see [55] for a comprehensive overview). These algorithms typically work in the configuration space (C-space) of the robot. Elements of the overall planning tasks are listed below [56].

- Representation of robot's configuration space.
- Representation of paths and trajectories.
- Kinematic or dynamic constraints.
- Sampling new points in the C-space.
- Measuring distances in the C-space.
- Interpolating between two points in C-space.
- Computation of forward or inverse kinematics.
- The global planner algorithm itself.
- Specification of start and goal conditions.
- Local planner for quickly connecting two configurations.
- Representation of the robot's geometry and the environment in the Cartesian 3D space.
- Collision checking.
- Updating path according to changing environments.
- Handling graph- or tree-like structures for roadmaps or discretization of the C-space.

In particular collision checking is known to play often a crucial role for sample-based planners, taking up most of the processing time.

In order to better understand the domain, an exhaustive research of the best practice open source libraries, which address many of the functionality mentioned above, has been performed. The next subsection reports the results of this survey while subsection 5.3.2 highlights the issues that make hard the interoperability between these libraries.

5.3.1 Open source libraries

The Motion Strategy Library (MSL)[57] was developed by the research group of Steven LaValle at the University of Illinois. The library includes support for multiple planners (including variants of RRT, PRM and Forward Dynamic Programming FDP), collision checkers (PQP) and visualization in multiple formats. Originally deployed only under Linux, a Windows version was published in 2008.

The Motion Planning Kit (MPK)[58] was developed by the research group of Jean-Claude Latombe at Stanford University. It implements a fast single-query bi-directional probabilistic roadmap path planner with lazy collision-checking (SBL) and relies on PQP for collision detection.

The Motion Planning Kernel (MPK)[59] was first developed by Ian Gipson at the Computational Robotics Lab at Simon Fraser University. It comes with a full suite of collision detection algorithms (V-collide and SOLID amongst others) and implements path planners for RRT and PRM.

Components for Path Planning (CoPP)[50] was developed by Morten Strandberg at the Royal Institute of Technology (KTH) with the aim of a clearly structured object-oriented planning framework. Several functionalities such as samplers, metrics, local planners, interpolators, and collision checkers (PQP, YAOBI) are explicitly distinguished with separate base classes. The library includes planners for PRM, RRT and PCD and provides support for visualization via Coin and VRML.

OpenRAVE [60] developed by Rosen Diankov is a framework that covers the whole development cycle around manipulation and grasping, including

support for different sensor inputs, controllers and physical simulation. Plugins are meant to provide an easy way for users to add various custom functionalities. OpenRAVE integrates to ROS and has interfaces to Octave, Matlab and Python.

The Lydia Kavraki's laboratory developed The Object-Oriented Programming System for Motion Planning (OOPSMP)[61]. It includes a large variety of motion planners and can handle kinodynamic constraints. Several general purpose data structures and functionalities for the domain of motion planning are provided.

Open Motion Planning Library (OMPL)[62] developed by Ioan Sucan stands out from the other libraries in the way that it explicitly concentrates on the core path planning algorithms. Other elements such as collision checking, simulation or motion control are handled by integration into the ROS framework. OMPL provides various planners including RRT, EST, SBL and KPIECE, and an inverse kinematics solver (GAIK) based on Genetic algorithms.

KineoWorks is the only framework mentioned here which is not available as open source. Originally developed as Move3D [63] at LAAS, it was put into a product by Kineo. It is meant to provide a component-based architecture that supports easy integration into applications. However it is not free and for this reason this section will not provide further information about it.

All of the libraries discussed above but KineoWorks are published as open source or are free for non-commercial use. Most of them are under active development, while only a few seem to be discontinued.

Notably all libraries above are written in C++. Some include scripting support for other languages or interfaces to tools such as Octave or Matlab. All of them offer some kind of 3D visualization, some also support for simulation and physics engines; with the special note of OMPL that relies on the ROS environment for all those aspects.

5.3.2 Interoperability issues

Most of the libraries cannot be easily interchanged and it is rather difficult to compare individual algorithms between libraries. One of the reasons is that they rely on some base classes, which sometimes are very detailed or include certain dependencies and that cannot easily be replaced or changed. Thus it is difficult to plug one algorithm, including all relevant aspects, into some other piece of software. In addition, many of the more internal aspects of planning algorithms, such as samplers or metrics, may not be made explicit. In order to exchange them, the algorithms' source code would have to be changed. The dependency of base classes on external frameworks may also restrict the transfer of a library onto real robots possibly with embedded PCs and limited resources. It should be noted that in particular the newer libraries include several mechanisms and design aspects that aim at minimizing the before-mentioned problems.

Nearly all libraries provide some kind of support for different implementations of functionalities, most commonly in the form of inheritance from a base class. This holds true for the main planner classes, but for example also collision checking engines are nearly always made explicit and interchangeable.

As an example, MSL is built around the classes `Model` (representing kinematic and dynamic systems), `Geom` (geometric objects for collision checking), `Problem` (general class to represent aspects of a path planning problem), `Solver` (base class for path planning algorithms), `Scene`, `Render` and `GUI` for visualization. New functionality may be added by inheriting from these main classes. In this specific library, metrics and interpolators for example may be changed by overriding virtual functions in the `Model` class. OpenRAVE and OOPSMP in addition provide the concepts of plugins, where different kinds of functionalities may be attached from outside. Those plugins have to inherit from base classes as well, but can then be loaded during runtime from dynamic libraries.

In OpenRAVE a central class `EnvironmentBase` glues all parts together. This is a container for all other elements, including physics and visualization. Most elements refer back to this container, e.g. loading from XML,

connecting robots with collision checkers, or drawing are handled via calls to `EnvironmentBase`.

In OOPSMP the composition and configuration of planning problems is done via customized XML files. A parser translates the XML elements into calls to dynamic libraries. In this way OOPSMP can be flexibly configured without touching any source code. On the other hand some kind of dedicated plugin functionality is needed to extend it. Similar to OpenRave, one container class `CoreRobotData` includes pointers to all components such as workspace, state space and smoother, with these components inheriting from `CoreRobotData`.

In contrast to the previous frameworks, OMPL is inherently integrated into ROS. The environment representation for the collision detector can be provided at runtime by an appropriate ROS node. The representation of the robot is loaded from URDF files. When paths have been planned, they are published to the ROS network. OMPL implements a number of abstract base classes such as `Planner`, `Path`, or `Goal`.

A major step concerning interoperability has been made in the ROS project. There a plenitude of standard interfaces for various aspects, from trajectories to robot kinematics and environment modeling, have been introduced in a data-centric way, without unnecessary functional dependencies. The libraries OpenRAVE and OMPL can be used over those interfaces, increasing the interoperability significantly.

The following tables provide an overview on how some of the key concepts for motion planning tasks are implemented in the different software libraries. These include data structures to represent *Configuration* (Table 5.1), *Path* (Table 5.2), *C-Space*, (Table 5.3), *Robot Kinematics* (Table 5.4), and functionality such as *Metric*, *Interpolator*, *Sampler* (Table 5.5), *Collision Detection*, and *Environment Modeling* (Table 5.6). Many of these concepts are represented in semantically similar ways. However the remaining differences constitute several of the major problems concerning interoperability between libraries.

Library	Main class	Notes
MSL	MSLVector	`double` array, includes size
MPK_{Kernel}	Configuration	`vector <double>`, includes calls to OpenGL
CoPP	Config	`vector <double>`
MPK_{Kit}	mpkConfig	`vector <double>`, includes various functions
OpenRave	TPOINT	`vector <dReal>`, includes, velocities and time
OOPSMP	State_t	`double` array
OMPL	State	`double` array

Table 5.1: Classes for point in C-space

Library	Main class	Notes
MSL	Planner	`list<MSLVector>`, located directly in planner's base class
MPK_{Kernel}	PA_Points	`vector<Configuration>`, includes calls to OpenGL
CoPP	Path	`list<Cinfigurations>`, includes time
MPK_{Kit}	sblPlanner	`list<mpkConfig>`, includes various functions
OpenRave	Trajectory	`vector<TPOINT>`, `vector<TSEGMENT>`, includes elements for dynamic motion control
OOPSMP	Path	includes interfaces for time, splitting and more. Base class with various implementations
OMPL	Path	points to a `SpaceInformation`, derived classes include array of `State`

Table 5.2: Classes for path or trajectory

Library	Main class	Notes
MSL	Problem, Model	includes upper/lower limits, start and goal configuration. control inputs and system simulation
MPK_{Kernel}	Universe, RobotBase	includes upper/lower limits, start and goal configuration
CoPP	DOF_Properties	stored in multiple places, where needed
MPK_{Kit}		limits are implicitly in planner
OpenRave	ConfigurationState	includes limits and number of DoF
OOPSMP	StateSpace	includes bounding box and various other functions, many concrete implementation
OMPL	SpaceInformation	includes start and goal configurations, dimension, `StateDistanceEvaluator`, `StateValidityChecker`

Table 5.3: Classes for C-space

Library	Main class	Notes
MSL	Model	includes kinematic structure and control
MPK_{Kernel}	RobotBase	`vector <LinkBase*>`
CoPP	KinematicNode	`vector <DOF_Properties>`, limits are stored in Robot class as well
MPK_{Kit}	mpkBaseRobot	includes pointer to a parent joint, spatial transforms, triangulated link model, PQP and SoQT data
OpenRave	KinBody	`vector vector<Joints> vector <Links>`
OOPSMP	StateSpace	implicitly defined via `StateSpace` and related classes
OMPL		based on ROS using URDF files

Table 5.4: Robot Kinematics data structures

Library	Notes
MSL	`Model` (and `Problem`) with virtual functions for Metric and Interpolator. Sampling as virtual function `ChooseState` in each planner class
MPK_{Kernel}	Sampler and metric as virtual functions in planner base classes. Interpolation hard-coded in planner
CoPP	classes `Metric`; `Interpolator`; `ConfigSpaceSampler`
MPK_{Kit}	non-virtual functions in class `mpkConfig` for metrics and interpolating. Sampling hard-coded in planner
OpenRAVE	classes `DistanceMetric`; `SampleFunction`; four interpolation methods hard-coded in `Trajectory`
OOPSMP	distance function in `StateSpace`; classes `PathGenerator`; `ValidStateSampler`
OMPL	classes `StateDistanceEvaluator`; `StateSamplingCore`; interpolation done in planners

Table 5.5: Interfaces for Metrics; Interpolator; Sampler

Library	Notes
MSL	`Geom` with derived class for PQP
MPK_{Kernel}	`CollisionDetectorBase`. `Universe` has an array of `Mesh` which can model various objects.
CoPP	`ObjectSet`. Base class `Geom` stores a position, with inherited classes for triangles and convex objects.
MPK_{Kit}	`mpkCollDistAlgo` uses PQP or own collision detector
OpenRAVE	`CollisionCheckerBase`. `KinBody` includes `TRIMESH` and `GEOMPROPERTIES` for modelling triangle meshes
OOPSMP	`CollisionDetector`. `Workspace` holds list of `Part`, support of polygons
OMPL	Based on ROS with interfaces of `CollisionSpace` and various geometry messages

Table 5.6: Interfaces for Collision detector and environment modeling

5.4 Case study - Harmonization

The harmonization of a class library such as CoPP has required the redistribution of responsibilities (i.e. functionalities) among the original classes, the design of new classes and the definition of well defined provided and required interfaces that make the functionalities available to the clients. In particular, the refactoring process of the CoPP motion planning library consisted of the application of the refactoring patterns introduced in the section 5.2.

The description of how the CoPP library has been refactored is organized in two subsections:

- subsection 5.4.1 explains how functionalities and data structure have been decomposed in a set of modules[1];
- subsection 5.4.2 describes how one of these modules has been designed.

5.4.1 Modules identification

The first step consisted of analyzing the data structures used in the various motion planning libraries (see subsection 5.3.2) in order to identify similar-

[1]In this chapter the term module is used for defining a set of classes that collaborate for providing a set of services. It is not an equivalent of the terms component or component framework used in the rest of the document.

ities. As expected, nearly all the algorithms rely on the concept of robot configuration, but the data structures used to represent them show subtle differences. While an array, or vector, of double values is used in all the cases, the differences concern additional information (e.g. time and flags, and whether the number of dimensions is included or not). For some versions it is trivial to convert between each other, while for others it is a problem due to different kind of information stored.

The refactoring pattern *Move Behavior Close to Data* suggests introducing data containers for harmonizing existing data structures and assigning them operations to be executed on the encapsulated data.

The guidelines of this pattern have led us to the definition of the new *ConfigurationSpace* module, which is depicted in figure 5.2 (modules are drawn according to the UML component notation where lollipops represent provided interfaces while sockets required interfaces). The *ConfigurationSpace* module is in charge of maintaining the representation of the robot configuration space. A *Configuration* is represented as a vector of n values, each one representing a point in the robot's configuration space. The *ConfigurationSpace* module stores at least the following configurations:

- *lowerConfig* is the configuration that represents the lower bound of the configuration space
- *upperConfig* is the configuration that represents the upper bound of the configuration space
- *currentConfig* is a list of robot's configurations at given instants of time.

Depending on the type of configuration space, additional data may be required for example to represent rotational joints or a rigid body moving in 2D space, resulting in $SE(2)$.

The *ConfigurationSpace* module offers services that were implemented as separated class hierarchies in the original motion planning libraries, such as the algorithms for configuration interpolating and for measuring distances between pairs of configurations. The configuration pair may be provided by

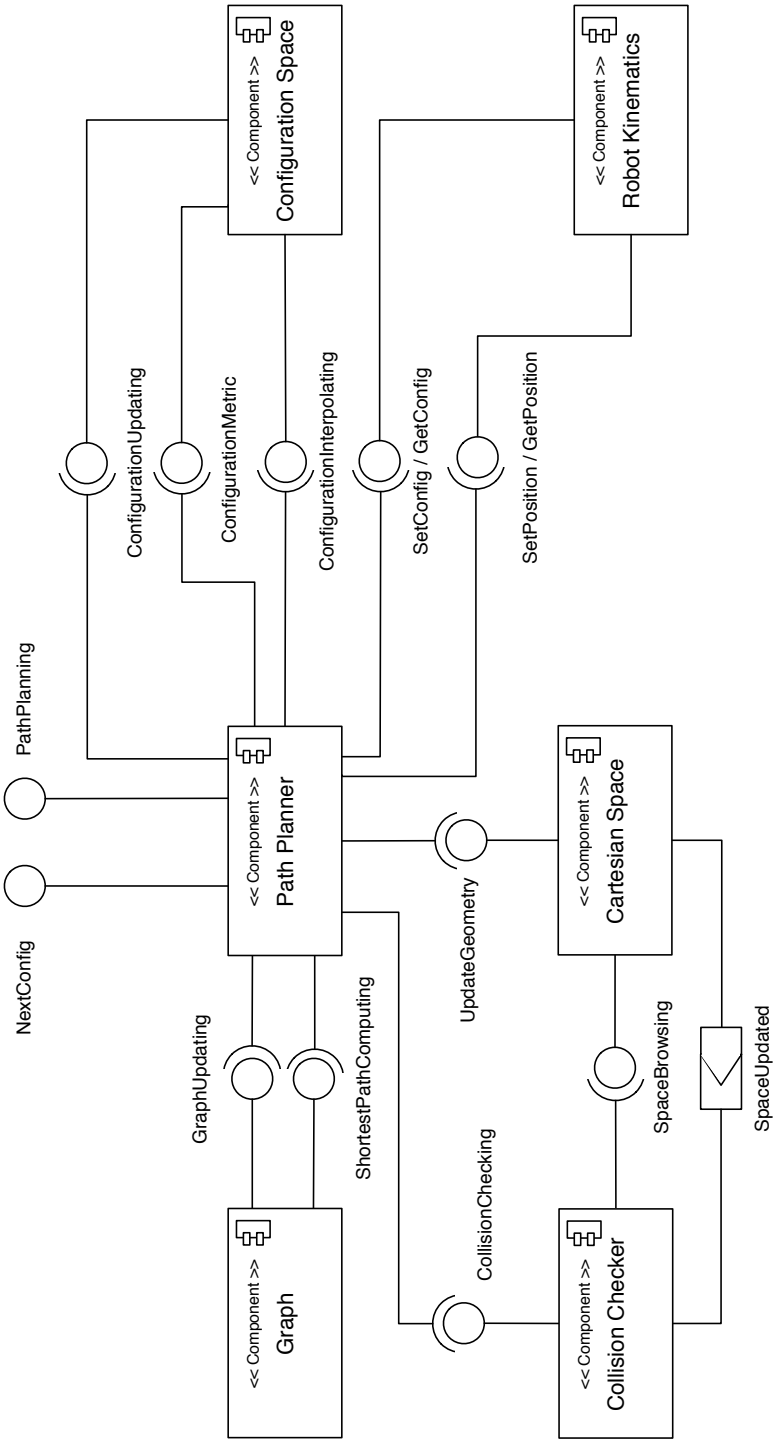

Figure 5.2: The system resulting from refactoring of motion planning library

the client or may correspond to two configurations at different instants of time. The following three provided interfaces have been defined:

- *ConfigurationSetup*
- *ConfigurationInterpolating*
- *ConfigurationMetric*

The *ConfigurationSpace* module does not implement any required interface. Thus it has no dependency to other modules and can be reused independently of other motion planning modules as building block for the implementation of robot functionalities where robot's configurations need to be represented, such as motion control, navigation and manipulation.

The refactoring pattern *Move Behavior Close to Data* has been iteratively applied to the original motion planning libraries. It has led to the identification of two more modules that behave as data container with provided interfaces only (see Fig. 5.2):

- The *CartesianSpace* module encapsulates the geometric representation of the robot's environment and implements interfaces for updating and browsing it.
- The *Graph* module is a wrapper of external graph management libraries that implements standard interfaces for creating, updating and processing data organized as graph structures.

Motion planning algorithms are often implemented by structuring the code according to the functional decomposition approach, where most of the logic of the functionality is provided by a single "god class", like *MotionPlanner*. The *Split up God Class* pattern refactors a procedural god class into a number of simple, more cohesive classes.

The iterative application of this pattern to the motion planning class libraries has generated three modules: *RobotKinematics*, *CollisionChecker*, and *PathPlanner*. The first iteration has produced the clear separation of two core functionalities, i.e. collision checking and path planning. Most class

libraries already offer distinct specialization hierarchies for the implementation of collision checking and path planning algorithms, but their high level abstract classes are incompatible and in some cases have a long list of methods with a large number of parameters.

The *CollisionChecker* module maintains an internal representation of the robot environment, which is an algorithm-specific approximation (e.g. using bounding boxes) of the Cartesian space. This internal representation needs to be updated when the robot's Cartesian space gets modified, for example when the robot or other objects change their position. For this purpose, this module requires the *SpaceBrowsing* interface of the *CartesianSpace* module and implements the *SpaceUpdated* event listener. The provided interface *CollisionChecking* defines the operations that the path planner can invoke to check and inspect collisions among objects in the robot's environment.

The *RobotKinematics* module stores the robot's kinematic model and implements only provided interfaces for invoking the forward and inverse kinematic transformations.

Finally, the *PathPlanner* module implements the algorithms that generate a robot path as a sequence of collision-free configurations. The simplified interaction between the seven modules in Fig. 5.2 consists of the following sequence of steps: *PathPlanner* generates (samples) a new robot configuration, updates *ConfigurationSpace*, gets the new robot position from the *RobotKinematics* module, updates *CartesianSpace*, checks if the new configuration is collision free and, if this is the case, updates the *Graph*.

Thus, it is clear that the *PathPlanner* module uses and integrates the services of the other modules to build a specific robot functionality and that these modules can be reused as building blocks for the implementation of other functionalities.

5.4.2 The Path Planner module

This subsection describes how the modules depicted in figure 5.2 have been refactored. In particular it illustrates the *PathPlanner* module, whose class diagram is depicted in figure 5.3. This module clearly separates stable data

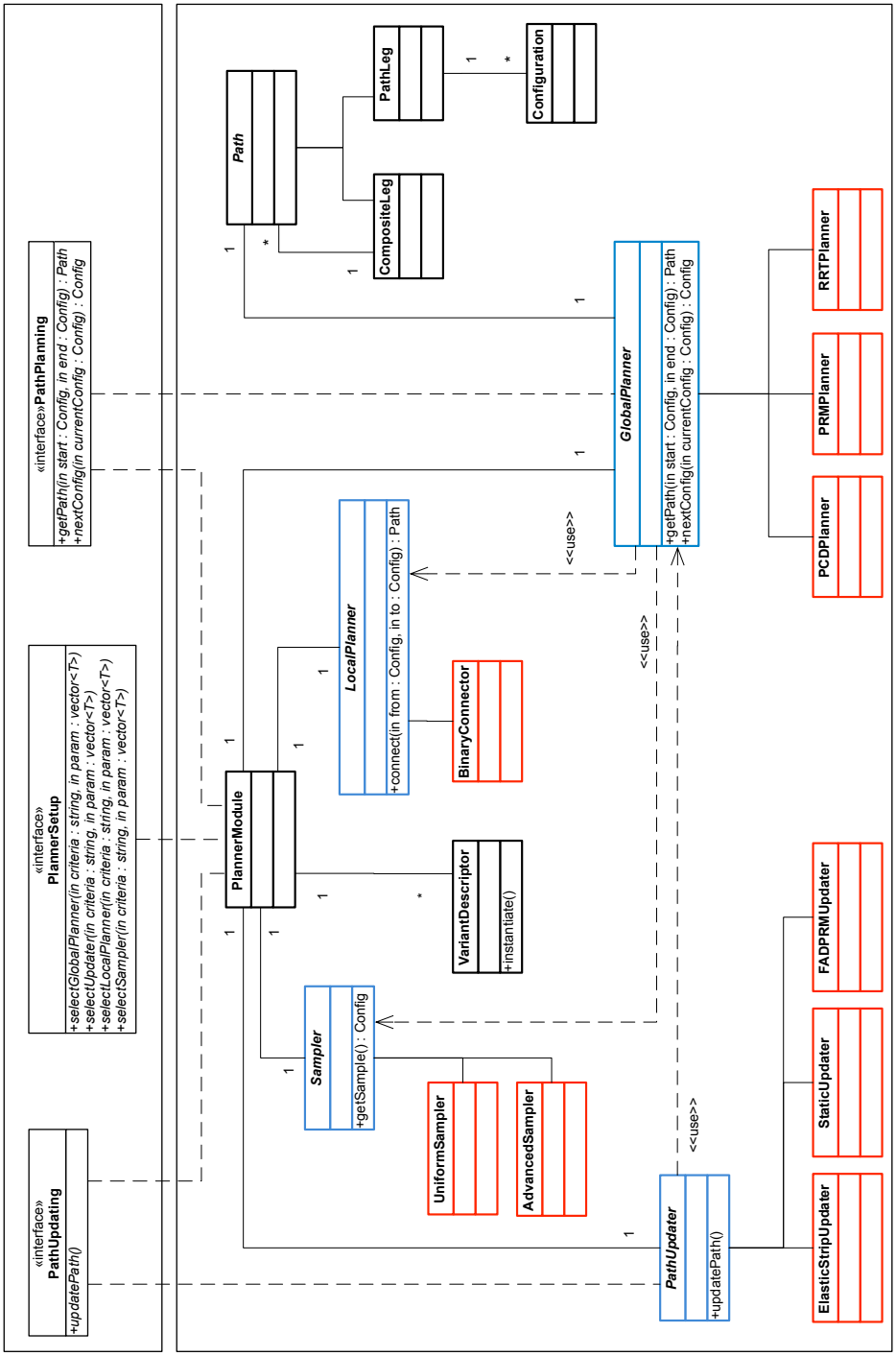

Figure 5.3: The Path Planner module

structures (black classes), variation points (blue classes) and concrete variants (red classes).

The classes used to represent a robot path are stable entities of the Path Planner module. They have been structured according to the *Composite* design pattern [53]. *PathLeg* is a sequence of *Configuration* objects. Taking as example a mobile manipulator that navigates inside a building, a path leg may correspond to the sequence of configurations of the mobile platform from a place inside a room to a place close to the door. Another path leg may then correspond to the sequence of configurations of the manipulator to open the door. *CompositeLeg* is a composition of path legs. The *Composite* design pattern allows the hierarchical composition of even more complex paths, which can be browsed through a uniform interface implemented by the *Path* abstract class.

Variation points and variants have been designed according to the Strategy Pattern. The variation points are abstract classes that implement stable data structures and operations that are common to a family of similar algorithms. From the analysis of motion planning libraries four core variation points have been identified: *GlobalPlanner*, *LocalPlanner*, *Sampler*, and *PathUpdater*.

The designed modules can be customized at design time, when the software developer defines concrete subclasses (e.g. *PRMPlanner*, *BinaryConnector*, *UniformSampler*, and *ElasticStripUpdater*) that implement specific algorithms and represent possible variants of each variation point (Strategy pattern). Another possibility is customizing the modules at run time, when one of several alternative variants is selected according to current execution context.

For example, variants of a specific family of algorithms could be switched through a graphical user interface in order to benchmark and compare their performance during experiment sessions. Alternatively, the robot could select the most effective algorithm autonomously according to situation awareness (e.g. a fast path planner in open space environments and a powerful path planner in cluttered environments).

Both situations require a module's client (e.g. the GUI or the robot controller) to switch among several variants. This is potentially implemented as long methods consisting almost entirely of case statements, which make

the code more difficult to maintain. As describe in the section 5.2, the pattern *Transform Conditionals into Registration* suggests removing the case statements by introducing a registration mechanism to which each variant is responsible for registering itself. The module clients are then in charge of querying the registration repository for retrieving references to the variants.

According to this pattern the class *VariantDescriptor* have been defined, which encapsulates the information necessary for registering, querying, instantiating and using each module variants. For each variation point (e.g. *GlobalPlanner*) the class *PlannerModule* encapsulates a member variable that points to the current selected variant.

Once the clients have customized the module by resolving its variation points, they need to access its functionalities, which in most of the cases are provided by the specific variant objects. For example, the *GlobalPlanner* variation point represents the core logic of the *PathPlanner* module and is available in several variants, each one implementing a specific algorithm for global planning. The *GlobalPlanner* and its variants implement interface *PathPlanning*, which defines the fundamental operation `Path getPath(Configuration start, Configuration end)`.

The analysis of the motion planning libraries described in section 5.3 reveals that clients of these libraries (e.g. the robot control application) typically have direct access to the objects that implement planning algorithms. In this case, direct access to variant objects would require navigating through classes *PlannerModule* and *VariationDescriptor* in order to get a reference to individual variant objects. This would violate encapsulation and would couple clients and variant objects unnecessarily.

Pattern *Eliminate Navigation Code* suggests preventing these problems by transforming object containers into service providers. This is the case of *PlannerModule*, which maintains pointers to current variant objects. It implements interface *PathPlanning* and delegates the execution of its operations to the current variant of the *GlobalPlanner*.

By applying the patterns *Transform Conditionals into Registration* and *Eliminate Navigation Code*, two distinct provided interfaces for the *PathPlanner* module have been defined. Interface *PlannerSetup* allows clients

configuring the module by selecting specific variants for each variation point. Interface *PathPlanning* allows clients accessing module's functionality. It is clear that different clients can access the two interfaces independently. For example, the GUI (one client) can switch two variants of the same variation point (e.g. *LocalPlanner*), which will be used to compute paths for the robot controller (another client).

6 A Product Line for Robust Navigation

This chapter describes how the development process presented in the chapter 3 has been applied to the Robust Navigation domain for defining a new Product Line, which allows the modeling and the resolution of the Robust Navigation variability.

6.1 The robust navigation

Robust navigation is the ability of a mobile robot to autonomously move from its current position towards a goal position, while avoiding collisions with unexpected obstacles (i.e. moving people) in an indoor environment such as a hospital or a museum. This task, which may seem simple, actually involves several functionalities:

- *Path Planning* computes an obstacle free path as a sequence of intermediate waypoints from the current pose to the target pose. This functionality requires a representation of the environment, which can be provided in different formats (e.g. a metrical map or grid-based map).
- *Pose Tracking* uses sensory data (e.g. wheels speed) to estimate the robot displacements and to update its current pose with respect to the initial pose.
- *Localization*, when a map of the environment is available, uses the

sensors data to estimate the robot current pose with respect to a global reference frame.

- *Trajectory Generation* receives as input the path computed by the planner and produces as output a trajectory. A trajectory is a refinement of the input path, which specifies linear and angular velocity for each one of its waypoints. These velocities are computed taking into account the robot kinematic and dynamic constraints and the task constraints (i.e. the robot is transporting a liquid container).

- *Trajectory Adaption* receives as input the generated trajectory and produces as output a modified trajectory that avoids unexpected obstacles detected by the robot sensors.

- *Trajectory Following* receives as input a trajectory and implements a feedback control loop that periodically reads the current robot pose and generates a twist (i.e. linear and angular velocities along the three axis) to minimize the distance to the path.

- *Robot Driving* receives a twist and generates velocity commands to the robot wheels. It produces as output the odometric estimate of the robot displacement from the initial pose by integrating the wheels velocities.

- *Laser Scanning* reads the raw data from a laser rangefinder and produces as output the corresponding measure expressed as a vector of points in polar coordinates (distance and heading).

From an algorithmic point of view, the challenging task is to organize an efficient interaction between these functionalities in order to maximize performance, safety, and robustness. Mobile robot navigation algorithms have been a research topic for several decades (see [64] for a taxonomy). Existing algorithms could be roughly classified as one- and multi-step methods. One-step methods directly convert the sensor data to a motion commands. Majority of one-step algorithms are either based on classical planning or on the potential fields approaches [64]. Today, they are rarely used due to their

inability to cope with dynamic environment and vehicle constraints. Multi-step methods (e.g. Dynamic Window Approach [30], Vector Field Histogram [31], Nearness Diagram [65]) overcome these limitations by creating a local map of the environment around the robot and performing local planning by computing possible motion directions (Nearness Diagram) and velocities (VHF) taking into account the distance to the goal or to a precomputed path.

From a software development point of view, the challenge is to implement robust navigation functionality as a set of reusable components that can be assembled into *flexible* systems. For this purpose, the development process described in chapter 3 has been applied, which avoids developing from scratch robot functionalities based on yet another software architecture. Instead, by collecting and analyzing well known open source libraries providing navigation functionalities, the process aims at identifying those architectural aspects (i.e. entities, data structures, interfaces, relationships) that are common to all or most of the implementations of the same family of functionalities, and those aspects that distinguish one implementation from another.

6.2 Open source libraries

This section presents the architecture of an open-source library, which provides mobile-based navigation functionality. The selected library was ROS, because of its popularity in robotic community (in particular the navigation stack coming with ROS Fuerte). Figure 6.1 shows a portion of the class diagram that represents the architecture of the ROS navigation stack. Below a brief introduction to the class library is provided and some drawbacks of this implementation are highlighted.

Class *BaseGlobalPlanner* is an interface of the global planners used in navigation stack. There are two implementations: *CarrotPlanner* and *NavfnROS*. The first one is a simplistic planner, which connects a target pose and the robot actual pose with a straight line and performs collision checking along this line. The second is a grid-based A* path-planner for circular robots.

Multiple functionalities are tightly coupled in the implementation of class *BaseLocalPlanner* (i.e. trajectory generation, adaptation and execution). It

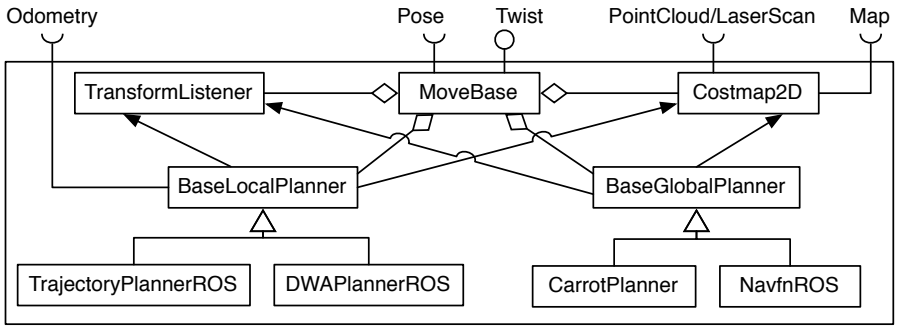

Figure 6.1: Mobile base navigation component in ROS

generates a number of trajectories for admissible linear and angular velocities of the robot. Each trajectory is scored according to an objective function, which includes goal heading, path heading and obstacle clearance. Finally the trajectory with maximum objective function is selected and its associated linear and angular velocity (twist) are sent to the robot driver. Two concrete implementations of this class are available, namely *TrajectoryPlannerROS* and *DWAPlannerROS*. Both assume implicitly that the robot has a differential drive kinematics model.

Class *CostMap2D* is an implementation of 2D occupancy grid-map. It embeds the data structures for representing a 2D tessellated representation of the environment. It is used for both path planning and obstacle avoidance.

The top-level class is the *MoveBase* class, which instantiates all the classes that implement specific functionalities and starts several threads for their concurrent execution. Concurrent access to shared resources (e.g. the map of the environment) is synchronized by means of infrastructure mechanisms, such as state machines, mutexes and numerous flags and conditions across the code. There is thus no guarantee that these functionalities are executed in real-time.

The *MoveBase* class is instantiated by a main function that starts a ROS node. Thus, all the functionalities for robust navigation are provided by a single component (ROS node). This component interacts with other components in the system (e.g. the robot base driver and the laser driver) by exchanging ROS messages. The set of exchanged messages represent

the component interface. Unfortunately, the component interface is not clearly separated by the component implementation since ROS messages are produced and consumed by several classes that implement the component. Thus, the only way to understand how components interact with each other is to carefully look at the source code.

The implementations of all the classes of the ROS Navigation stack are tightly coupled with the ROS infrastructure, thus they cannot be reused in a different environment.

6.3 Refactoring and component design

In order to improve the design of the open source library presented in the previous section, the refactoring patterns described in section 5.2 have been applied. The resulting classes have been then encapsulated into reusable components. The improvements of the new design are summarized in the following list.

- All the navigation functionalities are now mapped to finer grained classes, which can be encapsulated in components that have a single thread of control. This allows to replace individual functionalities easier and to select the most appropriate implementation for the specific task, environment and hardware.

- Accordingly, the trajectory generation, the trajectory adaption and the trajectory following functionalities are now implemented in three different components (in ROS there is a single node, which encapsulates the class *BaseLocalPlanner*). This separation reflects the different scheduling of the activities of the three components: the Trajectory Follower runs periodically at a higher frequency with respect to the Trajectory Adapter, as required by the closed loop position and velocity control algorithm. On the other side the Trajectory Generator runs aperiodically and computes a new output only when it receives an input path.

- According to the new design the trajectory adaption can now be performed only when strictly required, while previously it was performed continuously, also in the case of non approaching obstacles. Indeed the Trajectory Adapter component runs periodically, but it invokes the trajectory adaption functionality only when the information coming from the sensor indicates that the robot is approaching an obstacle.

- In its original design the BaseLocalPlanner provides as output a twist. It is the twist that, during the trajectory evaluation step, produces the trajectory with the maximum objective function (see the description of the BaseLocalPlanenr class in the previous section). However the produced twist is always applied with a certain delay (the time that goes from the start of the trajectory adaption functionality to the moment in which the twist is applied on the robot). This delay causes a discrepancy between the current pose of the robot and the pose of robot according to which the best trajectory was selected. As a result the applied twist drives the robot in a pose different from the expected ones. In the new design instead the Trajectory Adapter produces as output a Trajectory and the Trajectory Follower implements a position and velocity control that drives the robot in the desired pose by minimizing the position error.

6.4 Product line architecture modeling

The first part of this section describes a set of software architectures for Robust Navigation, which represent different products of the same product line. The objective is to show how the robotics variability influences the robot software architecture. Finally the second part introduces the model of the Robust Navigation Product Line.

Figure 6.2 depicts the architecture of a component based system that provides the subset of functionalities described in subsection 6.1. Here, *Boxes* represent software components. *Continuous lines* represent data flows or service invocation (according to the architecture Component-and-Connector

styles provided by the desired software framework). The line label describes the type of data exchanged between components. The dashed box groups components that together provide a high level robotic functionality (e.g. Local Navigation).

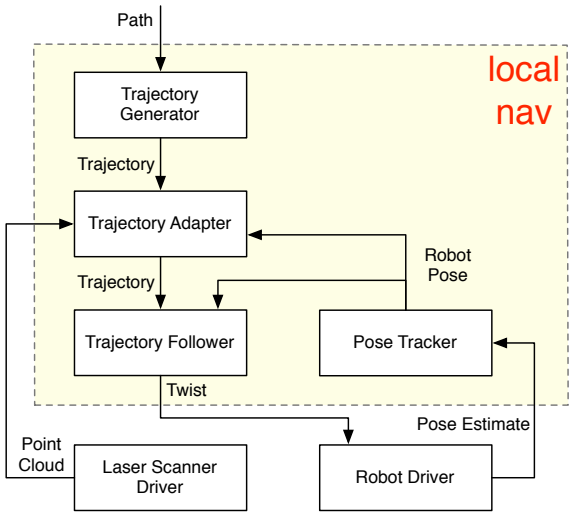

Figure 6.2: Local Navigation

The implementation of all the components in figure 6.2, but the *LaserScannerDriver* and the *PoseTracker*, depends on the robot kinematic and dynamic model (robot embodiment). Indeed, robot trajectories should be compatible with the robot kinematics and dynamics constraints. For example, a differential drive robot like the Pioneer 3-DX cannot move sideway while a robot with Ackerman steering cannot turn on place.

The implementation of the *Trajectory Adapter* is also influenced by the characteristics of the robot environment. For example, the VFH algorithm [31] has been proposed for fast robot traveling among densely cluttered obstacles, while the Elastic band approach [66] is adequate to avoid moving obstacles. The implementations of some of these algorithms have been refactored in order to harmonize their interfaces according to the architectures presented in this section.

Separating component interface from its implementation allows independent evolution of client and provider components. If client code depends

only on the interfaces to a component, a different implementation can be substituted without affecting the client code.

Clearly, there are constraints among the variants of each component implementation. For example, if the *Robot Driver* is specific for the differential drive robot kinematics, also the implementation of the other components should be specific for the same kinematic model. On the contrary, if the *Robot Driver* is specific for an omnidirectional robot kinematics (i.e. the robot can turn on place and move in every direction), the other components can implement algorithms that are specific for more constrained robot kinematics.

Robot intelligence, as the ability to express useful behaviors in the operational environment, is concerned with the navigation strategies for computing the path that the robot has to follow in order to reach the goal position. Depending on the environment, the task, and the available sensors, the robot can adopt two different navigation strategies for navigation: *deliberative* or *reactive*.

The *deliberative* strategy requires the robot to have a metric map of the environment, which specifies the position and shape of the surrounding obstacles (i.e. walls and furnitures) and allows the robot to identify the free space and plan obstacle-free paths. The map could be provided a priori or can be built by the robot itself during an exploration phase. This map-based navigation strategy requires the robot to have sensors (i.e. laser scanner or 3D depth camera) that provide accurate measurements of the surrounding obstacles for both map-building and map-localization. This strategy is convenient when the environment is mostly static (e.g. an ordinary home environment) and the task (i.e. floor cleaning) requires the robot to optimize resource usage (e.g. energy consumption) and meet specific constraints (i.e. the deadline for task completion).

The *reactive* strategy requires the robot to be able to recognize and follow landmarks in the environment. These could be natural landmarks, such as rocks on the Mars surface, or artificial landmarks, such as visual markers placed on the floor or on the walls of a hospital. In this case, the robot does not need a geometric map of the environment, as the paths are implicitly defined by the sequence of landmarks that the robot encounters while it is

navigating. The robot should be equipped with sensors (e.g. a monocular or stereo vision system) that provide information for estimating the relative position of the landmark with respect to the robot reference frame. This strategy is convenient when the environment can be structured in order to simplify the robot navigation stack. This is for example the case of industrial plants or public buildings.

The left-hand side of figure 6.3 depicts the Kuka youBot omnidirectional mobile robot performing a reactive navigation. In the figure it is possible to recognize a few visual markers placed on the floor. A laser rangefinder is mounted on the front side of the youBot robot at the height of the wheels. The right-hand side of figure 6.3 shows a screenshot of the ROS 3D visualization environment (Rviz). The detected markers are represented as red arrows, the green arrows indicate the local geometric paths followed by the youBot between subsequent markers, and the white polygonal lines represent the laser measurements of the walls around the robot.

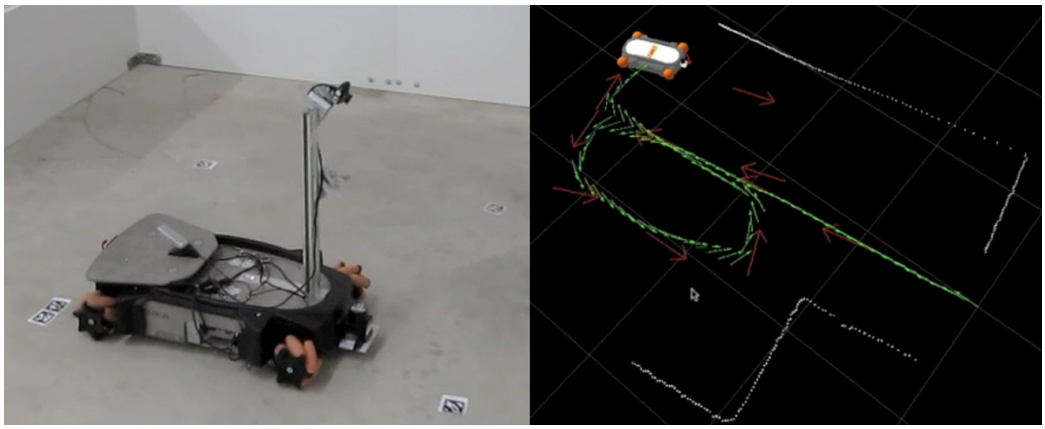

Figure 6.3: The youBot performing Marker Based Navigation

6.4.1 Map-based navigation

Figure 6.4 depicts the architecture for a *Map Based Navigation* strategy. The dashed lines indicate the new connections between the components for *Local Navigation* and the components for the *Map Based Navigation*.

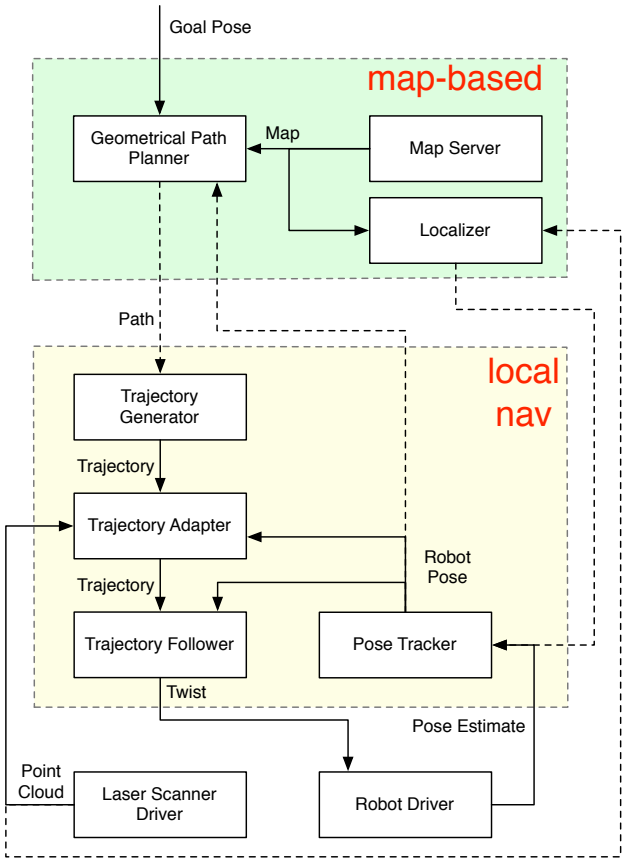

Figure 6.4: Map Based Navigation

The *Map Server* component is a data storage of 2D or 3D geometric maps. Several different representations of geometric spatial data have been proposed in the literature, which include for example the *Occupancy Grid* [67] and the *Polygonal Map* [68]. The *Map Server* component provides an interface that allows other components to retrieve the geometrical representation of the environment.

A large variety of path planning algorithms have been proposed (see [55] for a survey) for specific environment representations or for specific operational conditions (i.e. static or dynamic environment). Chapter 5 described how several path planning algorithms implementations have been refactored and packaged into class libraries that provide harmonized data structures and interfaces.

The map of the environment can be used also for estimating the robot pose with regards to the map reference frame. For this purpose, the *Localizer* components implements an algorithm that compare the measures provided by a sensor (e.g. a laser rangefinder or a 3D depth camera) with the environment representation in the map. Clearly, the localization algorithm depends on the type of map used to represent the environment. Thus, the implementations of the *MapServer* and of the *Localizer* must be compatible.

Another variation point regards the sensors used for the localization. The presented architecture assumes to use a laser scanner and requires the use of a *Localizer* implementation suited for this device. However other devices, such as a depth camera, can be used as well. In order to improve the flexibility of the architecture, the interface between the *Localizer* and the sensors has been designed by taking into account the sensor variability. In this way several sensors can be used without modifying the component interfaces. Indeed the *Point Cloud* is a data type that can be used both for laser scanners and depth cameras, because a laser scan can be basically considered a 2D point cloud.

6.4.2 Marker-based navigation

Figures 6.5 and 6.6 represent the architectures for *Marker Based Navigation*. Both robot embodiment and situatedness have an impact on these architectures.

The camera can be mounted on a support placed in a fixed position on the mobile robot (architecture depicted in figure 6.5) or can be attached to the arm of a mobile manipulator robot (architecture depicted in figure 6.6). In this second case the camera pose can change at runtime according to the arm configuration. This setup is needed for example when the camera should be able to locate markers that are placed not only on the floor but also on the walls around the robot.

The *Marker Locator* component processes the images received from the *RGB Camera Driver* in order to identify the unique ID of the visible markers and to estimate their position and orientation with respect to the robot reference frame. Several software libraries are available for implementing this

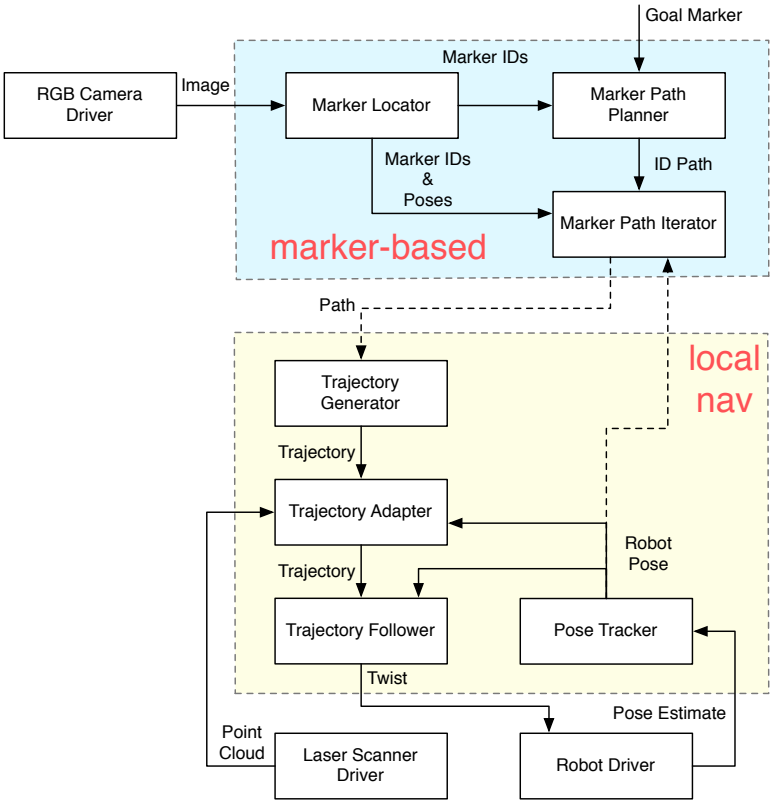

Figure 6.5: Marker Based Navigation with the camera in a fixed position

component, such as ARToolKit [69] and ARToolKit Plus [70]. Each library can be used to localize specific marker types.

In order to compute the position of a visible marker with respect to the robot reference frame, the *Marker Locator* component needs to know the camera pose with respect to the robot reference frame. When, the camera is mounted on the robot's arm (figure 6.6), the *Kinematics* component is needed, which receives the current position of the arm joints from the *Robot Driver* and sends the current camera pose to the *Marker Locator*. In contrast, when the camera is mounted on a fixed support (figure 6.5), its pose can be set as a component property and the *Kinematics* component is omitted.

The *Marker Path Planner* component periodically receives the IDs of the currently visible markers. When it receives also the ID of a goal marker, it generates a sequence of marker IDs (which is called ID path and is the shortest

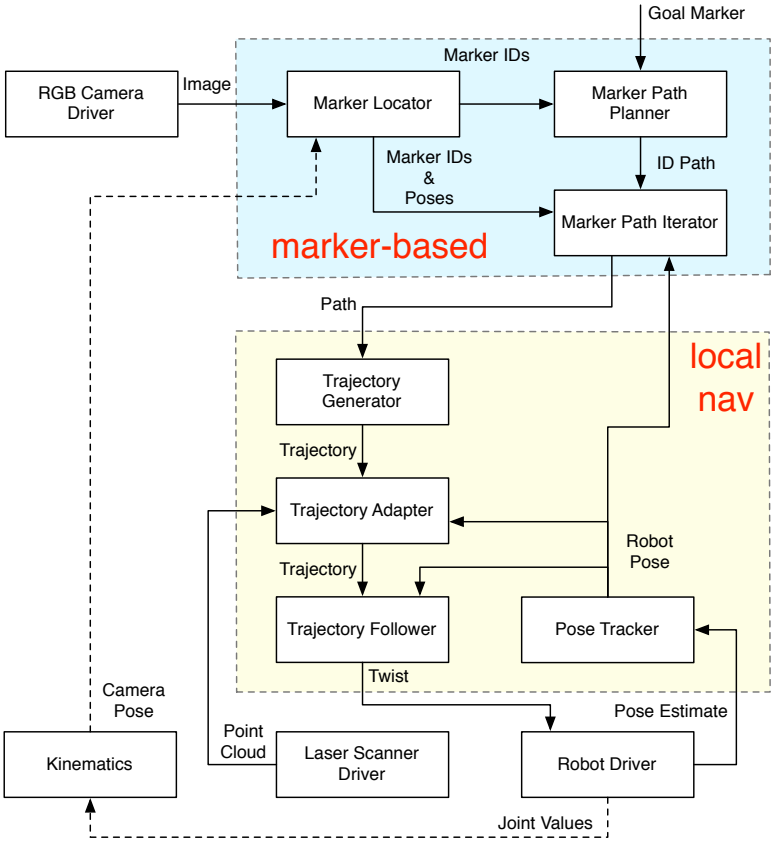

Figure 6.6: Marker Based Navigation with a moving camera

among the possible) from one of the visible markers to the goal marker.

The *Marker Path Iterator* component receives the IDs and poses of the visible markers, selects the next marker according to the ID path, and generates a geometric path containing only the pose of the next marker along the path.

In this case, despite the output path contains a single pose, the provided interface of the Marker Path Iterator has been designed by using the Path data structure (and not just a Pose) for improving the flexibility of the architecture. Indeed in this way a unique required interface of the *Trajectory Generator* can be used for both Map Based and Marker Based Navigation Strategies. Moreover other navigation strategies can be built on the top of Local Navigation components.

6.4.3 Hybrid navigation strategy

Let's consider a scenario where the robot navigates in an environment characterized by several separated buildings like an University campus. While it is inside a building the robot can navigate using a geometric map of each floor. Outdoor the robot follows the markers that indicate the paths among the buildings. For this purpose, the architectures for both *Map Based Navigation* and *Marker Based Navigation* need to be integrated. This is done by adding a *Global Planner* component, which specifies alternatively a *goal pose* or a *goal marker*.

Figure 6.7 depicts the resulting architecture. In order to simplify it some components and the relative connections have been omitted. They are connected in the same way described in the previous architectures.

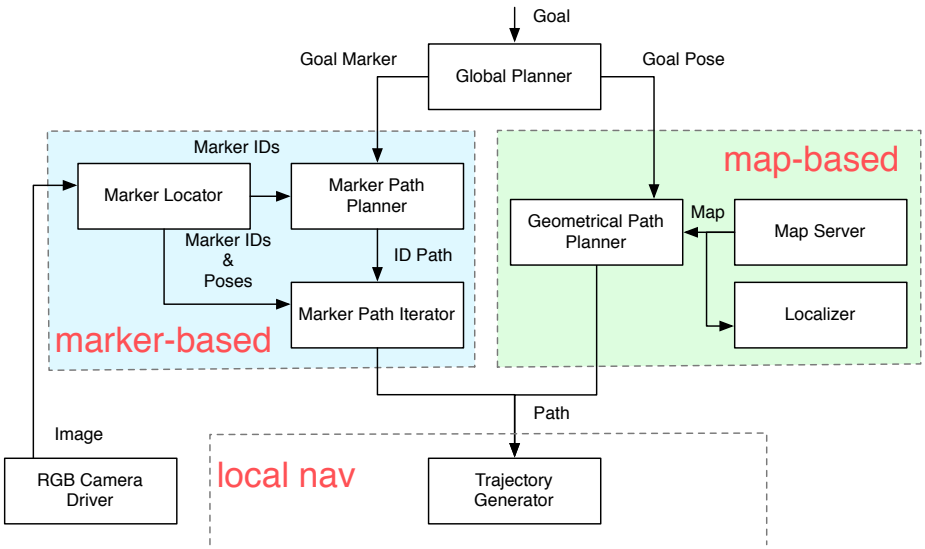

Figure 6.7: Map Based and Marker Based Navigation

6.4.4 The product line model

All the products, whose architectures have been described above, belong to the Robust Navigation Product Line. The model of the product line is depicted in figure 6.8. The model contains all the components used in the previous

architectures and the connections between components that are typically used together. Table 6.1 summarizes the components by highlighting their inputs and outputs, which ones are periodic and which ones are mandatory.

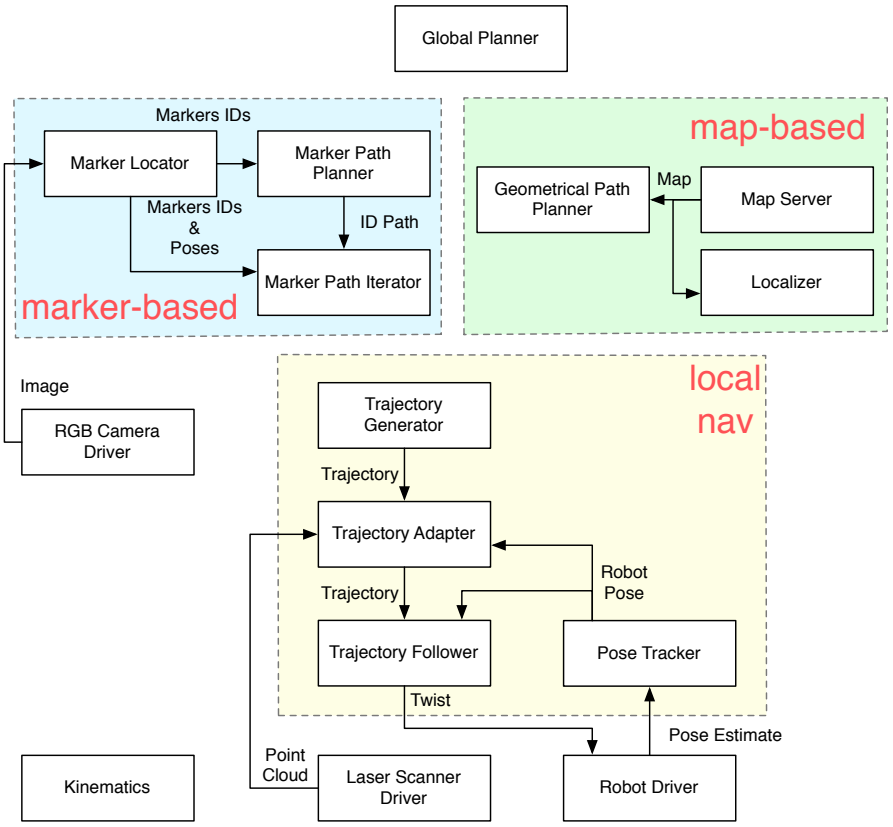

Figure 6.8: The Template System Model of the RN Product Line

Tables 6.2, 6.3, 6.4, 6.5 and 6.6 describe the interfaces of each component. In particular the data types are expressed by using the ROS messages IDL, which doesn't imply the use of ROS as software framework.

The objective was reusing as much as possible the existing messages. However for some interfaces, which required data structures not provided by the ROS messages, a new set of messages (*RN_msgs*) has been defined.

The massages *PoseStamped*, *Transform* and *Twist* are part of the package *geometry_msgs*; the messages *JointState*, *Image* and *PointCloud* are part of the package *sensor_msgs*; the messages *OccupancyGrid* and *Path* are part

Component	Req Interface	Prov Interface	Periodic	Mandatory
Trajectory Generator	Path	Trajectory	No	Yes
Trajectory Adapter	Trajectory, Robot Pose, Point Cloud	Trajectory	Yes	Yes
Trajectory Follower	Trajectory, Robot Pose	Twist	Yes	Yes
Pose Tracker	Pose Estimate	Robot Pose	Yes	Yes
Robot Driver	Twist	Pose Estimate, Joint Values	Yes	Yes
Laser Scanner Driver	—	Point Cloud	Yes	Yes
RGB Camera Driver	—	Image	Yes	No
Geometrical Path Planner	Goal Pose, Map, Robot Pose	Path	No	No
Map Server	—	Map	No	No
Localizer	Point Cloud, Map	Pose Estimate	Yes	No
Marker Locator	Image, Camera Pose	Marker IDs, Marker IDs & Poses	Yes	No
Marker Path Planner	Goal Marker, Marker IDs	ID Path	No	No
Marker Path Iterator	ID Path, Marker IDs & Poses, Robot Pose	Path	No	No
Kinematics	Joint Values	Camera Pose	No	No
Global Planner	Goal	Goal Marker, Goal Pose	No	No

Table 6.1: The Robust Navigation Product Line components

of the package *nav_msgs*; finally the message *String* is part of the package *std_msgs*.

The *RN_msgs* (*MarkerIDs*, *MarkerPose*, *MarkerPoses*, *Trajectory* and *Waypoint*) are reported in the listing 6.1.

```
Waypoint:
    geometry_msgs/Pose pose
    geometry_msgs/Twist twist

Trajectory:
    Header header
    Waypoint[] waypoints

MarkerIDs:
    int32[] IDs

MarkerPose:
    Header header
    int32 IDs
    geometry_msgs/Pose pose

MarkerPoses:
    MarkerPose[] poses
```

Listing 6.1: The implementation of the *RN_msgs*

Component	Data Port	Req/Prov	Event Port	Data Type
Trajectory Generator	Path	Req	Yes	Path
	Trajectory	Prov	—	Trajectory
Trajectory Adapter	Trajectory	Req	No	Trajectory
	Point Cloud	Req	No	PointCloud
	Robot Pose	Req	No	PoseStamped
	Trajectory	Prov	—	Trajectory
Trajectory Follower	Trajectory	Req	No	Trajectory
	Robot Pose	Req	No	PoseStamped
	Twist	Prov	—	Twist
Pose Tracker	Pose Estimate	Req	No	PoseStamped
	Robot Pose	Prov	—	PoseStamped

Table 6.2: The interfaces of the Local Navigation components

Component	Interface	Req/Prov	Event Port	Data Type
Robot Driver	Twist	Req	No	Twist
	Pose Estimate	Prov	—	PoseStamped
	Joint Values	Prov	—	JointState
Laser Scanner Driver	Point Cloud	Prov	—	PointCloud
RGB Camera Driver	Image	Prov	—	Image

Table 6.3: The interfaces of the Driver components

Component	Data Port	Req/Prov	Event Port	Data Type
Geometrical Path Planner	Goal Pose	Req	Yes	PoseStamped
	Map	Req	No	OccupancyGrid
	Robot Pose	Req	No	PoseStamped
	Path	Prov	—	Path
Map Server	Map	Prov	—	OccupancyGrid
Localizer	Map	Req	No	OccupancyGrid
	Point Cloud	Req	No	PointCloud
	Pose Estimate	Prov	—	PoseStamped

Table 6.4: The interfaces of the Map Based Navigation components

Component	Data Port	Req/Prov	Event Port	Data Type
Marker Locator	Image	Req	No	Image
	Camera Pose	Req	No	Transform
	Marker IDs	Prov	—	MarkerIDs
	Marker IDs & Poses	Prov	—	MarkerPoses
Marker Path Planner	Goal Marker	Req	Yes	String
	Marker IDs	Req	Yes	MarkerIDs
	ID Path	Prov	—	MarkerIDs
Marker Path Iterator	ID Path	Req	Yes	MarkerIDs
	Marker IDs & Poses	Req	Yes	MarkerPoses
	Robot Pose	Req	No	PoseStamped
	Path	Prov	—	Path

Table 6.5: The interfaces of the Marker Based Navigation components

Component	Data Port	Req/Prov	Event Port	Data Type
Kinematic	Joint Values	Req	Yes	JointState[]
	Camera Pose	Prov	—	Transform
Global Planner	Goal	Req	Yes	String
	Goal Marker	Prov	—	String
	Goal Pose	Prov	—	PoseStamped

Table 6.6: The interfaces of the remaining components

6.5 Variability model

The feature diagram depicted in figure 6.9 captures the functional variability of the Robust Navigation Product Line. For sake of simplicity the figure does not represent the variability regarding all the different implementations of the components and the different models of sensors.

The feature diagram indicates that the *local navigation* functionality is mandatory and that at least one navigation strategy has to be selected.

Two alternative robot kinematics model are supported, i.e. the *omnidirectional* or the *differential drive*. The camera can be mounted in a *fixed* or *movable* support.

Two alternative algorithms are available for both the *Trajectory Adaptation* (Dynamic Window Approach [30] and Vector Field Histogram [31]) and the *Geometric Path Planning* (Probabilistic Roadmap and Rapidly-exploring Random Trees [55, chap. 5]) functionalities.

It should be noted that the *Geometrical Path Planning* feature could have been modeled as children of the *Map* navigation strategy and the *Camera Position* feature as children of the *Marker* navigation strategy. Instead, the feature model is enriched with two constraints:

- *Map* REQUIRES *Geometrical Path Planning*
- *Marker* REQUIRES *Camera Position*

The functionalities for Robust Navigation are provided by software components that have been implemented according to two robotic-specific component models (*ROS* and *Orocos*), which are represented as two alternative features.

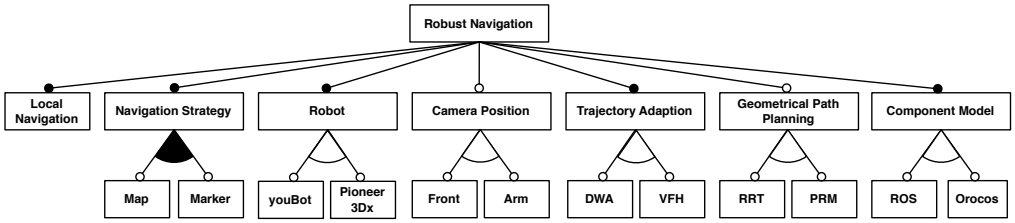

Figure 6.9: The Feature Model of the Robust Navigation Product Line

6.6 Variability resolution model

The product line presented in the previous sections allows the deployment of different applications. This section describes how the required elements and the transformations map the functional variation points (Feature Model) and the architectural elements (Product Line Model).

The mandatory feature *Local Navigation* represents a stable point and defines as required elements the components for the local navigation, the *Robot Driver*, the *Laser Scanner Driver* and the connections between all of them (i.e. these components and connections are present in all the applications of the product line).

The variation point related to the selection of the navigation strategies involves the connection transformation. When only the feature *Marker-Based* is chosen, then the connections between the *Marker Based Navigation* components (including the *RGB Camera Driver*) and the *Local Navigation* components have to be created. The required elements of the *MarkerBased* feature are the *Marker Based Navigation* components and the connections between them. Hence the *Map Based Navigation* components, the connections between them and the *Global Planner* can be removed from the model because they are not required by any feature.

When only the *MapBased* feature is selected instead, the *Map Based Navigation* components have to be connected to the *Local Navigation* components. In this case the *MapBased* feature defines as required element only the *Map Based Navigation* components, hence the *Global Planner*, all the components for the *Marker Based Navigation* and their connections are not required by any feature and can be removed.

When both the features are selected all the connections described above and the connections to the *Global Planner* have to be created. Moreover the *Global Planner*, the *Marker Based Navigation* and the *Map Based* components have to be conserved because they are all required elements.

The variation point regarding the camera involves a connection transformation and a property transformation. When the camera is in a fixed pose (feature *Fixed*) a property transformation is used for setting the value of the

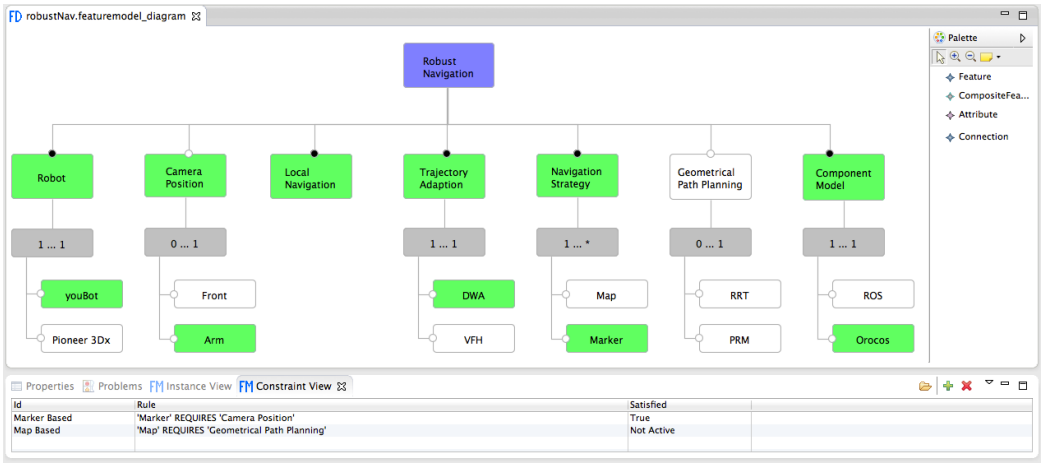

Figure 6.10: A screenshot of a feature selection

Marker Locator property (it defines the pose of the camera with respect to the robot reference frame). When the camera is mounted on the arm instead (feature *On the Arm*), a connection transformation is used for connecting the kinematics component (which provides information about the camera pose) to the *Robot Driver* and the *Marker Locator*. To be noted that the feature *On the Arm* is the only one that defines the *Kinematics* component as a required element. Hence when this feature is not selected the *Kinematics* component is always removed from the model.

The variation points regarding the Robot Trajectory Adaption and the Geometrical Path Planning involve some implementation transformation. According to the selected features different implementations for the *Local Navigation* components and the *Geometrical Path Planner* have to be used.

Finally the software framework variation point is only used by the resolution engine (see subsection 4.5.1) for deciding which Template System Model has to be used in order to generate the Configured System Model.

6.7 An example of product derivation

Figure 6.10 depicts a screenshot of the Feature Selector tool, which shows a Feature Model Instance for the robust navigation product line.

Given this feature selection and the Product Line Model depicted in figure

6.8, the resolution engine will produce as output a Configured System Model describing the Architecture for the Marker Based Navigation with a moving camera, which is depicted in figure 6.5.

Part II

Approaches for designing flexibility systems

7 JOrocos

Recent advances in robotics and mechatronic technologies have stimulated expectations for emergence of a new generation of autonomous robotic devices that interact and cooperate with people in ordinary human environments.

Engineering the control system of autonomous robots with such capabilities demands for technologies that allow the robot to collect information about the human environment, to discover available resources (physical and virtual), and to optimally exploit information and resources in order to interact with people adequately. Common approaches in robotics build on sophisticated techniques for perception and learning, which however require accurate calibration and extensive off-line training.

Recent approaches investigate how the robot can exploit the World Wide Web to retrieve useful information such as 3D models of furniture [71] and images of objects commonly available at home [72]. In [73] the *Robotic Information Home Appliance* is illustrated as a home robot interconnected to the home network that offers a friendly interface to information equipment and home appliances.

This new trend poses new challenges in the development of robot software applications since they have to integrate robotic and information systems technologies, which account for quite different non-functional requirements, namely performance and real-time guarantees at one side and scalability, portability, and flexibility at the other side.

Modern robot control systems are typically designed as (logically) dis-

tributed component-based systems, where the interactions between components (control, sensing, actuating devices) are usually more complex compared to more traditional business applications. In Robotics, the software developer faces the complexity of event-based and reactive interactions between sensors and motors and between several processing algorithms. For this reason, robotic-specific component-based models and toolkits have been developed, which offer mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management, and system configuration.

In contrast, the most common middleware infrastructures for the World Wide Web and home networks are the Java Platform Enterprise Edition and Service Oriented Architectures. Service Oriented Architectures (SOA) have been proposed as an architectural concept in which all functions, or services, are defined using a description language and where their interfaces are discoverable over a network [74].

Some attempts to develop robotic applications as SOA systems can be found in the literature (a recent survey can be found in [75]). Their main disadvantage is that they give up the typical component-based nature of robotics systems and force a pure service oriented approach. More recently, Service Component Architectures (SCA) [24] have been proposed as an architectural concept for the creation of applications that are built by assembling loosely coupled and interoperable components, whose interactions are defined in terms of bindings between provided and required services. As such, SCA offer the advantages of both the Component-based engineering approach typically used in robotics and the Service Oriented Architectures.

In order to bridge the gap between current component-based approaches to robotic development and modern information systems technologies, the JOrocos library, which extends the popular Orocos robotic framework [23] with Java technologies, has been developed. Thanks to JOrocos, a robot control application can be designed as a SCA system, where components encapsulating real-time control functionality are seamlessly integrated with web services and the most common Java toolkits, such as the SWING framework for developing graphical user interfaces.

7.1 SCA - Orocos integration

In order to make possible the interaction between SCA and Orocos components some architectural mismatches presented by the two frameworks had to be faced. In fact, despite both SCA and Orocos components interact by exchanging messages, the syntax and semantics of these messages is fundamentally different.

In SCA messages are used for invoking services provided by components. Services are defined by explicit interfaces that completely describe the name of each operation, its arguments and the return value (the signature of the method). The message sent by the requester component to the provider component describes which operation has to be executed and provides its parameters. Hence the execution of the component functionality starts when the message is received.

In Orocos instead the communications are based on data flows and the messages are used for exchanging data. Components periodically elaborate data received on the input ports and write their results on the output ports. This means that the components business logic is regularly executed every T milliseconds, where T is the period of the component.

This meaningful difference introduces two main problems:

1. How an invocation of a SCA service can produce an input that will be processed in the next cycle of an Orocos component business logic?

2. How the data published on an Orocos output port can trigger the execution of a SCA service?

Let's introduce how these problems have been solved by means of a simple scenario in which two SCA components and an Orocos component cooperate in order to move a Kuka youBot [76] towards a given position. The youBot is a mobile manipulator with an omnidirectional and holonomic base and a five degrees of freedom arm. The components are described below:

- A SCA component, called *Locomotor*, which provides a service for moving a youBot towards a position defined by the client and monitoring

its activity. The component is in charge of transforming the given cartesian position in a set of commands (joint positions), forwarding them to the robot and retrieving its status. In order to do that the component requires two services, which are provided by the driver of the robot for sending and receiving this information.

- An Orocos component, called *youBot Driver*, which provides an input port and an output port. The component implements the API of the youBot and is in charge of actuating the axes in order to reach the joints positions specified by the client on the input port. The output port is instead used for periodically publishing the status of the robot, for example the position and the velocity of the joints.

- A SCA component, called *SCA youBot Driver*, which is implemented by using the JOrocos library and represents a proxy to the youBotDriver component within the SCA system.

The *SCA youBot Driver* is described by means of the following interfaces:

- Provided interface *sendingCommand*. This interface provides a service for receiving commands from the *Locomotor* and writing these commands on the input port of the *youBot Driver*. The result is the activation of the *youBot Driver* operations that are in charge of moving the robot.

- Provided interface *retrievingStatus*. This interface is invoked by the *Locomotor*. The *SCA youBot Driver* periodically checks and retrieves the new data available on the *youBot Driver* output port. The interface provides the operation for retrieving these values.

- Required interface *notifying*. This interface is provided by the *Locomotor* and is used by the *SCA youBot Driver* for notifying some events. The possible events are *idle*, *busy*, *refresh*. The component raises the event *busy* when it starts the execution of an operation and the event *idle* when this operation is completed. In this way the *Locomotor* knows whether the operation that it requested is completed or not. The event

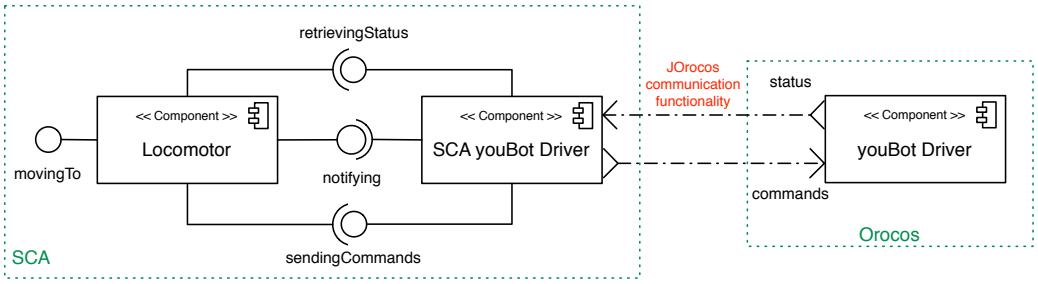

Figure 7.1: The youBot Scenario

refresh is instead raised when new data are read from the *youBot Driver* output port.

Here it is important to consider that, in order to notify the *Locomotor* about the availability of new data before the *youBot Driver* deadline, the *SCA youBot Driver* should check the output port with a frequency at least two times greater than the one of the Orocos component.

The components and the interfaces of this scenario are depicted in figure 7.1.

Another difference between SCA and Orocos regards the synchronization of the component operations after the action of sending a message. In SCA it is possible to define, by means of an annotation, whether the message is sent in a synchronous or asynchronous way. In the first case the thread that sends the message suspend itself until the result is returned. In the second case instead the thread continues its execution without waiting for the return value. A callback message will notify the component when the return value of the sent message will be computed. In Orocos all the messages are sent in an asynchronous way. The components read the data on the input ports and publish data on the output ports without waiting for other component activities.

JOrocos faces this problem by providing the possibility of reading data from the Orocos output port in an asynchronous way, according to the Publish/Subscribe communication paradigm [77] (more information about the implementation will be described in the subsection 7.1.3). In this way both SCA and Orocos component don't have to wait after sending a mes-

sage. However a synchronous communication can always be defined in the implementation of *SCA youBot Driver*.

The last mechanism provided by SCA that is not defined in Orocos is the hierarchical composition of the components. In SCA this functionality is available by means of the concept of composite. A composite contains different components and allows the developer to promote a set of their services in order to make them accessible to the clients of the composite. In this way a composite can be reused as a simple component in a more complex architecture.

Here it is possible to leverage on this SCA mechanism and create composites that contains different bridges to Orocos components (like the *SCA youBot Driver*) and promotes their operations as services. This approach is inspired by the Facade design pattern, which aims to provide a unified interface to a set of interfaces in a subsystem [53]. In this way a single reusable SCA component, the composite, can provide to its clients the functionalities defined in several Orocos components.

7.1.1 The JOrocos library and its architecture

The JOrocos library offers a set of mechanisms that allow the implementation of the proxies of Orocos components mentioned in the previous pages (e.g. *SCA youBot Driver*). These mechanisms provide the functionalities for reading and writing on Orocos data ports, reading and writing Orocos properties and invoking operations provided by Orocos components.

Another interesting mechanism offered by JOrocos is the introspection of Orocos running components. It provides the functionality for discovering at runtime which components are available, their ports, their operations and their properties. This mechanism allows the development of systems more complex than the scenario defined in the introduction of this section: systems in which the SCA composite doesn't have a priori knowledge of the Orocos components and configures itself at runtime according to the information retrieved through the introspection. For example, with reference to the previous scenario, it will be possible to design a system in which the SCA

composite doesn't know at compile time which robot has to be controlled. This information will be retrieved at runtime by introspecting the current *Robot Driver* component and according to its ports the *SCA Robot Driver* will configure itself.

The functionalities provided by JOrocos are realized on the top of Corba, the middleware that Orocos uses for exchanging messages between distributed components. Corba doesn't guarantee the respect of real-time constraints and for this reason when the communication between Orocos components has to be real-time the components have to run on the same machine. In this way the communication between the local components doesn't rely on Corba and so the respect of the real-time constraints is not compromised. In this direction the use of Corba is not a problem for the SCA-Orocos integration because the real-time components will be implemented in Orocos and will run on the same machine.

The architecture of the library is depicted in the UML class diagram reported in figure 7.2. As showed in the diagram the classes of the library are organized in two main packages: *core* and *corba*.

- The core package contains the classes that store data structures and offer operations that are middleware independent. These classes define the core of the library and represent the main entities of an Orocos system.

- The corba package contains instead the classes whose methods provide a set of operations that are corba specific.

The classes of the core package whose name starts with the word *Abstract* are abstract classes and have to be extended in order to provide the functionality that are middleware specific. They represent proxies of Orocos entities and offer methods for introspecting them and interacting with them. The other classes of the package are instead completely middleware independent.

The idea is that the separation of the middleware-independent parts (core package) from the middleware-specific parts (corba package) will allow in future an easier extension of the library in order to provide a support for other middlewares.

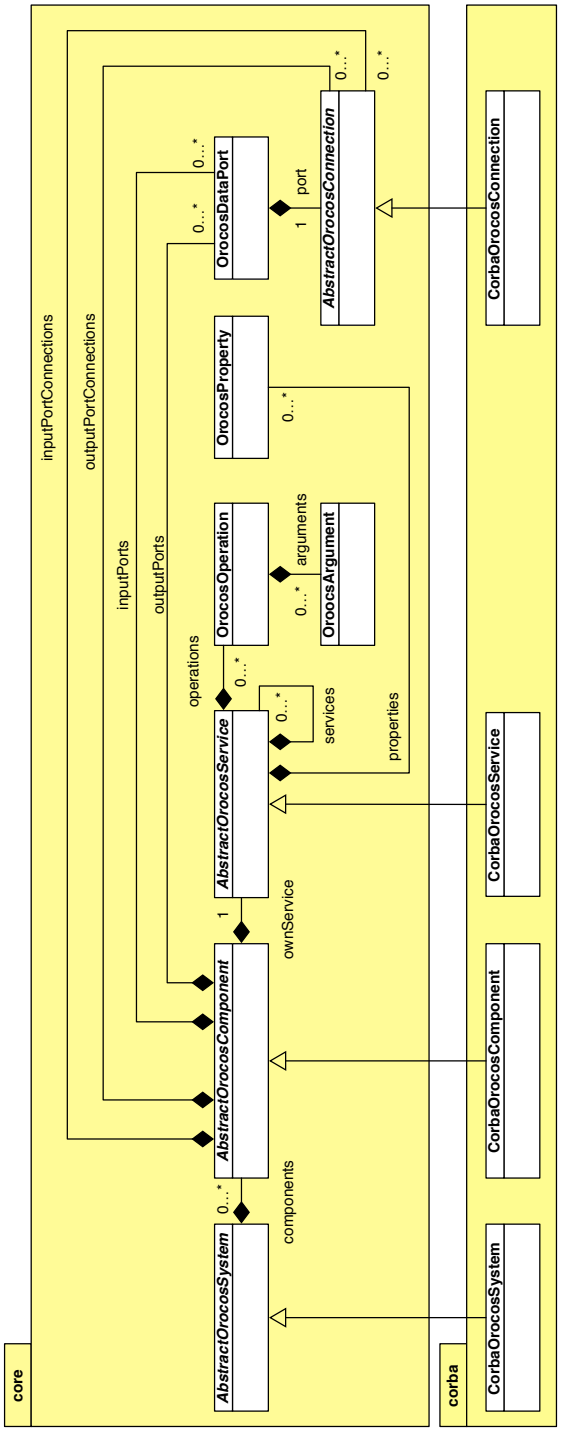

Figure 7.2: The JOrocos Architecture

The main class of the library is named *AbstractOrocosSystem*. It offers the functionality that allows a developer to connect his software to a running Orocos system, introspect its components and retrieve references to them.

An *AbstractOrocosComponent* is a proxy to an Orocos component and allows the clients to introspect its data ports and its own service. The class offers the operations for creating connections to Orocos ports and writing and reading data on these ports.

The data ports of an Orocos component are represented by means of the class *OrocosDataPort*. The interaction with these ports is made available by the class *AbstractOrocosConnection*, which provides the channel that allows the operations of writing data on the output ports and reading data from the input ports.

An *AbstractOrocosService* is a proxy to an Orocos service and offers the functionality for introspecting and invoking its operations and introspecting, reading and writing its properties. Orocos operations are typically not used for implementing the kind of operations that regard the business logic of the real-time components, but for example for configuring their period or retrieving information about their status (stopped, running, etc.). On the other hand properties are part of the Service Configuration interface and are used to load or tune application specific configurations at runtime (e.g. the parameters of a PID).

The operations of an Orocos service are represented by means of the class *OrocosOperation*. The properties are instead described by means of the class *OrocosProperty*.

7.1.2 The SCA-Orocos component

This subsection will explain how the interaction between Java and Orocos works and how the component *SCA youBot Driver* is implemented. The code reported in the listing 7.1 shows the interfaces of the services provided by the component. The annotation *@Callback* defines the interface that will be used for notifying the events to the *Locomotor*. The annotation *@OneWay* instead means that the invocation of method will be asynchronous.

```
1 public interface retrievingStatus{
2   public double[] getJointsPositions();
3 }
4 @Callback(Notifying.class)
5 public interface SendingCommands {
6   @OneWay
7   public void setJointsPositions(double[] values);
8 }
```

Listing 7.1: The interfaces of the *SCA youBot Driver* services

The listing 7.2 reports the variables declared in the implementation of the *SCA youBot Driver* component.

```
1  @Service(interfaces={retrievingStatus.class,SendingCommands.class})
2  public class SCAYouBotDriver implements retrievingStatus,SendingCommands,Observer{
3    @Property
4    protected String orocosIP;
5    @Property
6    protected String orocosPort;
7    @Callback
8    protected Notifying locomotor;
9    private AbstractOrocosSystem orocosSystem;
10   private AbstractOrodocComponent youBotDriver;
11   private double[] jointsPosition;
```

Listing 7.2: Part of the implementation of the *SCA youBot Driver* component

The class implements the interface *java.util.Obsever* (Observer design pattern [53]), which defines a method for being notified when a new data is available on an Orocos output port. Furthermore the class implements the two interfaces that describe the services.

The first row is a SCA annotation that defines the interfaces of the services of the component. The rows from 3 to 6 declare two SCA properties used for configuring the IP address and the port number of the Corba name service, rows 7 and 8 instead declare a reference to the SCA callback interfaces.

7.1.3 Read and write data on Orocos data ports

The JOrocos library allows both the operations of reading and writing on an Orocos data port. In order to be executed these operations require a

connection between the java client and the Orocos port. Two types of connections are available: data and buffer. On a data connection the reader has access only to the last written value whereas on a buffer connection a predefined number of values can be stored.

The listing 7.3 reports the constructor of the class *SCA youBot Driver* in which the connections to the port are created.

```
1 public SCAYouBotDriver(){
2     orocosSystem = CorbaOrocosSystem.getInstance(orocosIP,orocosPort);
3     orocosSystem.connect();
4     youBotDriver = orocosSystem.getComponent("youBotDriver", false);
5     youBotDriver.createDataConnectionToInputPort("commands", LockPolicy.LOCK_FREE, this);
6     youBotDriver.subscribeToDataOutputPort("joinstStatus", LockPolicy.LOCK_FREE, this, 500);
7 }
```

Listing 7.3: The implementation of the *SCAYouBotDriver* constructor

- rows 2-3 retrieve a reference to an Orocos running system and create a connection to it.
- row 4 retrieves a reference to the *youBot Driver* component.
- row 5 creates a data connection to the input port "*commands*" of the Orocos component *youBot Driver*.
- row 6 creates a data connection to the output port "*status*" and starts a thread that periodically checks if new data are available on the port. This functionality is implemented in JOrocos (methods *subscribeToDataOutputPort* and *subscribeToBufferOutputPort*). In this case a data connection with a lock free policy is created. The third parameter specifies the Observer object that will be notified when new data will be available on the port (in this case the component itself). Finally the last parameter defines the frequency with which the availability of new data on the port will be checked (it is expressed as period in milliseconds).

Once the component is subscribed to the output port it will be notified as soon as a new data will be available by means of the method *update* (inherited

from the Observer interface). The implementation of this method is reported in the listing 7.4. It simply stores the new data on the variable *jointsPosition* and notifies the *Locomotor* that new data are available.

```
1 public void update(Observable arg0, Object arg1) {
2   OrocosPortEvent event = ((OrocosPortEvent)arg1);
3   jointsPosition = ((YouBotStatus)event.getValue()).getJointsPosition;
4   locomotor.notify("refresh");
5 }
```

Listing 7.4: The implementation of the method *update*

At this point the *Locomotor* is able to retrieve the new data through the operation provided by the interface *retrievingStatus*. Its implementation is reported in the listing 7.5. It simply returns the position of the joints.

```
1 public double[] getJointsPositions() {
2   returns joinstPosition();
3 }
```

Listing 7.5: The implementation of the interface *retrievingStatus*

The listing 7.6 reports instead the implementation of the operation defined in the service *sendingCommands*. The purpose of this operation is writing the data received from the *Locomotor* to the Orocos output port. The component first notifies the *Locomotor* that the operation is started, then writes the values on the "*commands*" port and finally notifies the *Locomotor* that the operation is completed.

```
1 public void setJointsPositions(double[] values) {
2   locomotor.notify("busy");
3   youBotDriver.writeOnPort("commands", values, this);
4   locomotor.notify("idle");
5 }
```

Listing 7.6: The implementation of the interface *sendingCommands*

The operations of writing and reading data support both simple and complex data types and respectively receive as parameter and return as result instances of the class *Object*. In this context corba introduced two issues:

1. Corba returns references to the requested objects as instances of the class *Any* (*org.omg.CORBA.Any*). Hence the result of a read operation is an *Any* object. However the objective was returning a more general *Object* instance (*java.lang.Object*).

2. The cast from *Any* to the right type is possible only by means of the "*Helper*" classes that are automatically generated through the *IDL-to-Java* compiler. However it was not possible to know every possible data type a priori and consequently implement all the possible cast in the code of JOrocos.

These problems have been solved by means of the Java reflection. Indeed, in the code of the write and read operations the class name is retrieved from the object that has to be written (in the case of the write operations) or from the *Any* object (in the case of the read operations). Then the name of the class is used for loading at runtime the right "*Helper*" class and using its static method for casting *Any* to *Object* or vice versa. The listing 7.7 shows how JOrocos casts an *Object* to an *Any*.

```
1 // value is the "Object" that has to be written and has to be cast to "Any"
2 String className = value.getClass().getName(); + "Helper";
3 Class<?> helper = Class.forName(className);
4 Method castMethod = helper.getMethod("insert", Any.class, value.getClass());
5 // the insert method inserts value in the any object received as second parameter
6 castMethod.invoke(null, any,value);
```

Listing 7.7: The *Object* to *Any* cast

7.2 The case study

In order to test the functionalities provided by JOrocos a simple case study application have been implemented. It is similar to the scenario introduced in the section 7.1 but the *Locomotor* component is replaced by a graphical interface (*youBot Monitor* component), which is in charge of plotting the current state of the joints and allowing the user to set the period of the *youBot*

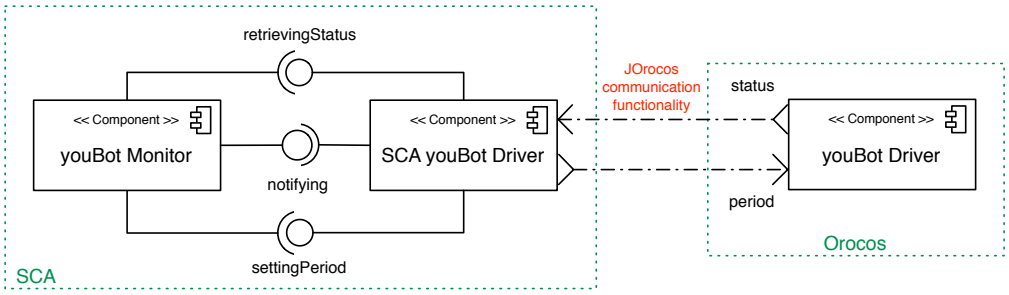

Figure 7.3: The case study components

Driver component (the inverse of the frequency with which the operations of the component is executed). The architecture is depicted in figure 7.3.

The *youBot Driver* component publishes on the output port a set of values that describe for each joint the actual position, velocity, current, temperature and error flag (10 bits that provide information about a set of possible errors). The component also has a new input port called *period*. When a new data is written on this port the component set its period according to the received value. Due to this new port also the *SCA youBot Driver* has a new provided interface named *settingPeriod*. It provides a service for receiving a new period value from the *youBot Monitor* and writing it on the input port of the *youBot Driver*.

The *youBot Monitor* component has two required interfaces that correspond to the provided interfaces of the *SCA youBot Driver*. It also provides the *notifying* interface, which is used by the *SCA youBot Driver* for notifying its events.

The SCA components and the Orocos component run on two different machines: the last one on the embedded pc of the robot whereas the other two on the supervisor workstation.

The implementation of the *SCA youBot Driver* component is very similar to the one reported in section 7.1. The *youBot Monitor* component is instead implemented by using the Java SWING and provides several tabs. In the main tab (figure 7.4) a set of global information about the state of the joints is showed. This tab also allows the configuration of the *youBot Driver* period. The other tabs (figure 7.5) instead provide information about a specific joint

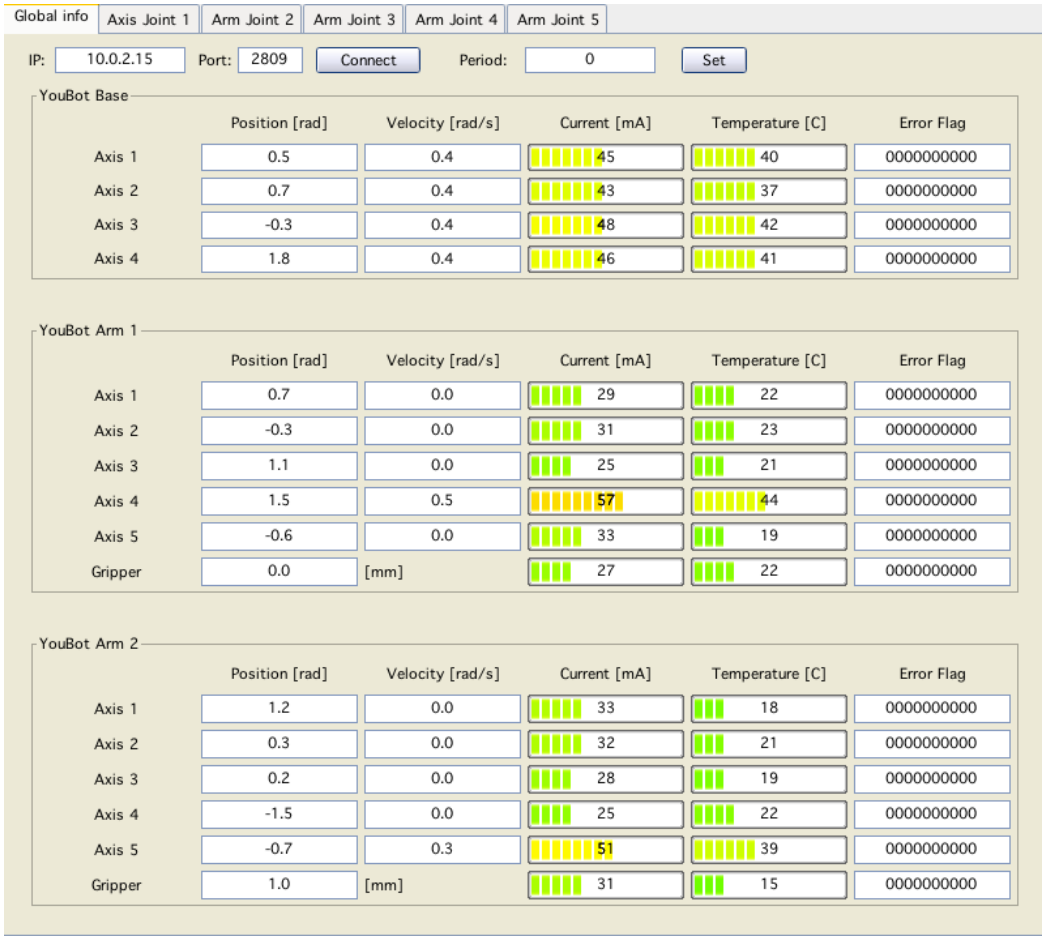

Figure 7.4: The tab reporting the global information

and plot on a set of charts the trend of the joint values.

This case study demonstrated how JOrocos makes possible and simple the communication and the cooperation between SCA and Orocos. By writing few lines of Java code it was possible to retrieve data from the Orocos output port and set the period of the *youBot Driver* component. Furthermore the location of the components over the network (except for setting the IP address and the port of the name service) and the different programming language used for implementing the Orocos component didn't represent a problem, because these issues were hidden by Corba.

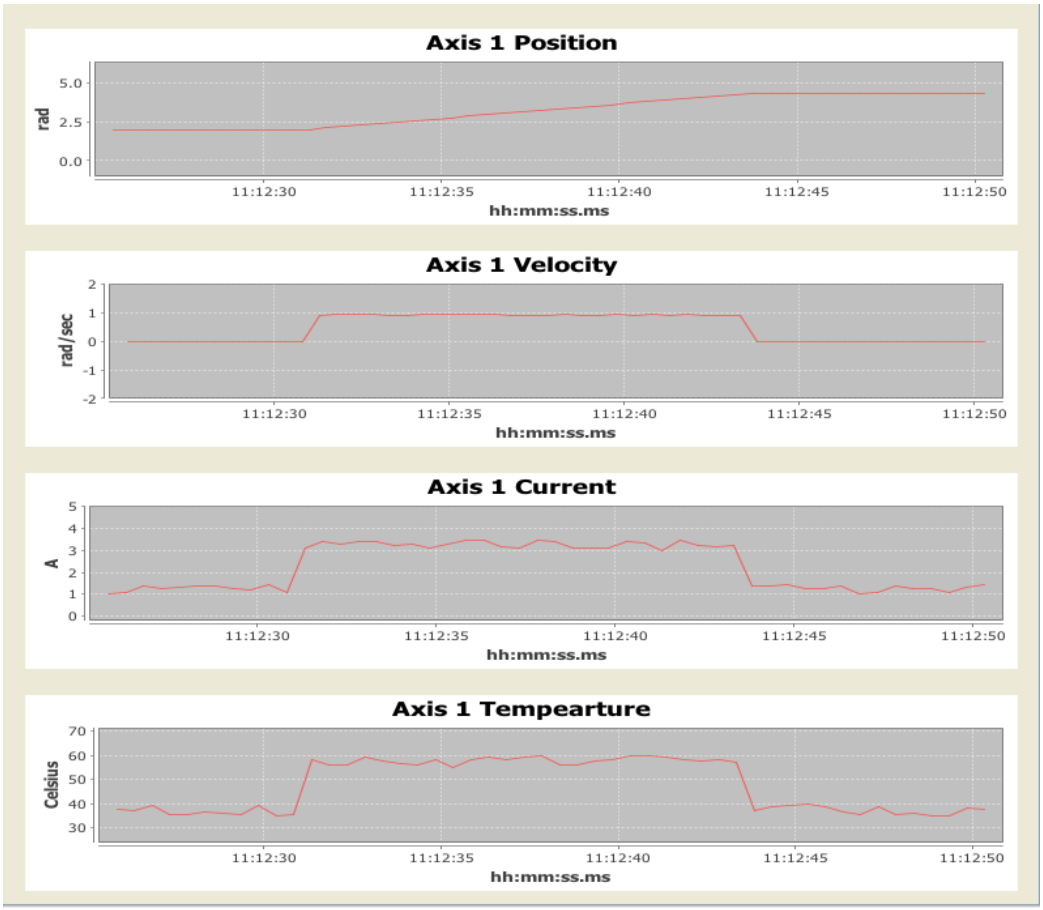

Figure 7.5: The tab reporting the trend of the joint values

7.3 Discussion ans future works

This chapter has discussed the problem of making possible the cooperation between Service Oriented Architectures (SOA) and Data Flow Oriented Architectures in the robotics field. In particular it focused on SCA and Orocos, the first a component based SOA and the second a hard real-time component based robotics framework. The chapter has presented a set of architectural mismatches between the two component models and a java-based library, named JOrocos, which allows the developers to bridge these differences by defining proxy components. Finally a set of guidelines for the development of these proxies have been presented and applied in a case study.

The first mismatch regarded the syntax and the semantics of the messages exchanged between the components in the two frameworks. Here JOrocos provides to the developers the mechanisms for allowing the communication between SCA and Orocos components and translating SCA messages to Orocos messages and vice-versa. However JOrocos doesn't provide the possibility of directly connecting a SCA Service (or Reference) to an Orocos Port. The developer has to define, according to the guidelines presented in this chapter, a proxy component that provides input to the Orocos component when one of its services is invoked and invoke a service of its client (the SCA component) when the Orocos component produces data on the output port. In this direction a possible improvement will consist of (a) using JOrocos for extending the SCA runtime in order to define a new binding for Orocos and (b) extending the SCA composite designer for supporting this new binding. These extensions will replace the role of the proxy component and will allow the developer to directly connect SCA and Orocos components.

The second mismatch was about the synchronization of the component operations after the action of sending a message. Here JOrocos doesn't provide the possibility of choosing a specific synchronization mechanism. In order to permit both synchronous and asynchronous way, a specific synchronization mechanism has to be implemented in the proxy component. For example, the scenario proposed in subsection 7.1.3 hs demonstrated how it is possible to send messages synchronously and asynchronously. Indeed the

operation of retrieving the robot status is executed by the *SCA youBot Driver* in a synchronous way, whereas the operation of sending commands in an asynchronous way.

Finally the last mismatch concerned the absence of a hierarchical composition mechanism in Orocos. Here JOrocos allows the developers to leverage on the SCA composition mechanism for encapsulating several Orocos proxies in a SCA composite and reusing it in complex and hierarchical systems.

8

Programming Languages Performance Comparison

Robot software systems are concurrent, distributed, embedded, real time, and data intensive. Computational performance is a major requirement, especially for autonomous robots, which process large volumes of sensory information and have to react to events occurring in the robotics operational environment.

In order to meet performance requirements, robotics algorithms have been typically implemented in C and C++. Robotics developers in fact have always considered C++ significantly faster than Java. Despite that, the idea of using it in robotics is not really new: it has been followed in several projects (see section 8.1) and recently Willow Garage and Google have started a project for developing a Java-based porting of ROS [78].

In this chapter a study on the comparison of performance between Java and C++ is reported. The goal is to quantify the differences and to offer a set of data in order to better understand whether the performance of Java allows to consider it a valid alternative to C++ or not, at least for non-real-time functionalities. For this purpose a well known algorithm originally written in C++ has been implemented in Java and then compared to the original one. The chosen algorithm is the Delaunay triangulation and its implementation comes from the OSG library [79]. It was developed in the computer vision field but it is typically used also in robotics for reconstructing environment surfaces from a set of 3D points. The algorithm is well suited for the purpose

of this study because it stresses several critical points of the programming languages performance such as: (a) the frequent access to the memory for operating on dynamic size array (massive use of the garbage collector) and (b) the frequent evaluation of logical conditions.

Although in the computer science domain many comparison studies have been proposed, this test can be considered interesting because the algorithm has been implemented and executed with a newer and improved version of the Java JDK. Indeed the current Java Virtual Machine (JVM) offers a new compiler, which greatly improves the performance of Java with respect to the older versions.

8.1 Java for robotics

Java is an object oriented programming language and it was intended to serve as a new way to manage software complexity. It offers to its users a set of software libraries and specifications, which allow the designing and the deploying of cross-platform applications. Java is used in different application domains such as enterprise resource planning (ERP) and web servers (e.g. JSP). It is widely spread also on mobile phones and embedded devices. This section presents a set of robotics projects developed with Java and a survey on several performance comparisons between Java and C++.

8.1.1 Robotics Java projects

During the 2011 Google I/O the researchers of Willow Garage and Google presented a new project that aims to develop a pure Java implementation of ROS [78]. By means of this project Google and Willow Garage aim to boost the development of advanced Android applications for robotics and easiness the access to the cloud computing for reducing the cost of the robotics hardware.

In [80] the integration of Matlab in a distributed behavioral robotics architecture is presented. The architecture is completely implemented in Java and leverages on the Jini platform for distributed object registration, lookup

and remote method invocation. The Matlab integration is realized by means of JMatLink and allows the invocation of Matlab scripts and the access to the Matlab workspace as a distributed object. The authors present as case study a multi-robot mines detection. In [81] a team from Lund University demonstrated that it is feasible to develop a motion control system entirely in Java. They designed an application that takes a picture of a person and controls a pick and place robot in order to draw on a paper the result of the shooting. The software and the motion controller guarantee the respect of the real time constraints by means of Java RTS. In [82] a real-time system for controlling a remote manipulator over a local area network or over internet is presented. The developers implemented both the control system and the teleoperation of the robot in Java. In [83] an autonomous motion planning system completely developed in Java is introduced. The application allows the user to set up the working environment though a graphical interface and offers the functionalities of collision detection, obstacle avoidance, free-paths generation and selection of the shortest path. Finally in [84] an application for controlling robots through the World Wide Web is implemented. The software is designed for dealing with low bandwidth and high latency and allows the operator to control the robot from any computer connected to the web.

8.1.2 Java versus C++

One of the main differences between Java and C++ is that the first was born as an interpreted language while the second as a compiled language. Compiled languages are translated into machine code trough a compiler. This process generates a file that can be directly executed by the CPU. Interpreted languages are compiled in a platform independent language (bytecode), which can be executed only by means of an interpreter (e.g. JVM). Hence, programs written in C++ (compiled language) are platform dependent and must be compiled for every computing platform before the first execution. Java programs instead are translated into bytecode only once and can be used on different platforms. However they have to be interpreted at every execution

(the JVM is platform dependent). For this reason interpreted languages are in general more flexible and portable than compiled languages but at the same time slower.

In order to improve the performance, in 1998 Java 1.2 was released with a new feature called Just-In-Time compiler (JIT)[85]. JIT is integrated into the JVM and is in charge of translating the Java byte code into binary code. Each method is translated only when it is called for the first time. Thanks to this improvement the execution time decreases and the code is again portable.

Many comparisons between C, C++ and Java were documented in literature. From this point the description calls "*Java*" the version optimized with JIT and "*interpreted Java*" the original version. In [86] the execution times of C++ and Java are compared. The authors tested the execution of four sorting algorithms, two of $O(n^2)$ complexity (bubble sort and insertion sort) and two of $O(n \cdot log(n))$ (recursive quick sort and heap sort), on four integer data sets of different sizes. The results demonstrated that C++ was much faster than pure interpreted Java (from 11 to 20 times) and only from 1.45 to 2.91 times faster than Java (version 1.3). In [87] a set of polynomial multiplications was computed and executed using the three languages. The results showed that Java completed the operations faster than standard C (mean of 21%) but in average 2.61 times slower than C++. In [88] the executions of the Linkpack benchmark were compared for Java and standard C. This benchmark was introduced by Jack Dongara and measures how fast a computer solves a dense N-by-N system of linear equations. The results showed that for a 1000 x 1000 system Java was 2.25 times slower than C. In [89] Ruolo evaluated the Java method call performance. Different tests with a different numbers of parameters showed that Java was only one clock cycle slower than C++. The same tests also highlighted that the time needed for allocating user defined objects on the heap was roughly equivalent. However C++ also uses the stack for allocating temporary object and in this case it was from 10 to 12 times faster than Java, which uses only the heap. Interesting conclusions were reported by Mangione [90]. He tested the repetitive execution of simple operations like integers and float divisions and showed that Java was as fast as C++. As summarized in table 8.1 all the papers report that, since the

Paper	Test	Java vs. C	Java vs. C++	JDK	C/C++ compiler
[86]	Sorting alg.	—	1.45 - 2.91	Sun 1.3	Borland v. 5.5
[87]	Polynomial mult.	0.79	2.61	Sun 1.2b5	Sun Workshop C 4.2
[88]	Linkpack bench.	2.25	—	Sun 1.2b4	—
[89]	Method call	—	1 clock slower	Please refer to the paper	
[90]	Int and float div.	—	~ 1	Sun 1.1.5	Visual C++ 5.0

Table 8.1: Results summary (Columns 3 and 4 report the execution time ratios)

introduction of the Just-In-Time compiler, Java is only 1.45-2.91 times slower than C++.

Since these studies demonstrated that the execution of simple operations in Java is more or less as fast as in C++, one factor that could influence the total execution time of a Java program is the Garbage Collector (GC). However [91] showed that Java GC is as fast as a *malloc/free* operation in C++. In fact when a program executes a *malloc* operation, the allocator looks for an empty slot of the right size and returns a pointer to a random place in the memory. In Java instead the allocator selects the bits of memory adjacent to the last bit it used. Hence it doesn't need to spend time looking for memory. In conclusion the amount of time used for the garbage collector is comparable to the amount of time that the allocator uses in C++ for finding free memory slots.

Finally other interesting results are documented in [92]. The same program was implemented by 40 different programmers in different languages (24 in Java, 11 in C++ and 5 in C). The experiment compared not only the performance of the languages but also the differences between the implementations in the same language (interpersonal differences). The results demonstrated that Java was 2 times slower than C++ and that the interpersonal differences were much larger than the average difference between Java and C++. That means a well written Java program can be as efficient as an average C++ program.

8.2 The performance comparison case study

Visual sensors such as laser scanners and depth cameras acquire information on the environment geometry in form of a point cloud: a set of vertices in a 3D coordinate system. Each one of these vertices corresponds to a point on the surface of one of the objects present in the environment. In order to reconstruct the surface of these objects the vertices have to be connected. This problem is called mesh generation and one of the possible solutions consists of the Delaunay triangulation [93].

Delaunay's algorithm connects the set of points in such a way to build a series of triangles that respect the following property: for all the set of points there is no point which lies inside the circumcircle of any triangle. The triangulation result is unique except if more than three vertices stand on the same circumference. In this case more than one solution exist.

In this section a comparison between a C++ and a Java version of the Delaunay triangulation will be reported. The Java version is the result of the refactoring of a C++ implementation coming from the OSG libraries [79]. The implementation of this triangulation algorithm is based on the Bowyer-Watson method, which works in the plane space. It iterates all the points of the cloud and for each one executes two main steps: identifying the triangles whose circumcircle contain the current analyzed point and then building a new set of triangles, which respect the Delaunay condition. This algorithm allows to process point clouds in 3D space but realizes only a triangulation in the plain space therefore the Z coordinate is ignored. It should be noted that the implemented algorithm does not provide a constrained Delaunay triangulation. For this reason, during the timing and the comparison of the computation time, the constraints of the OSG version have been excluded.

Both the OSG and the Java implementations receive as input the point cloud in form of a collection of vertices. The OSG implementation defines a custom class, *Vec3Array*, which is a specialization of the class *MixinVector*[1]. *Vec3Array* defines a vector of *Vec3* instances, which are triplets of float

[1] *MixinVector* allows inheritance to be used in order to easily emulate derivation from *std::vector* but without introducing undefined behavior through violation of virtual destructor rules [94]

data types. The refactored implementation instead uses the Java *ArrayList*. This collection has been chosen because it is the fastest of all the collections provided by the Java framework for what regards the operations of inserting, iterating and sorting [95], and because its performance are comparable with the one of Java *Vector*. On the other side *ArrayList*, like C++ *std::vector*, is not as well efficient when it has to perform the operation of removing elements in random position. In order to better understand how much the overhead between Java and C++ is due to these data structures, the performance of the *Vec3Array* and *ArrayList* collections have been compared by executing a set of tests on the operations that are most used during the triangulation algorithm.

- Insertion. This test executed 10000 and 100000 insertions of objects (instances of class that represent the 3D points) at the end of the two collections. The values of 10000 and 100000 have been chosen because they are the maximum orders of magnitude of the collection sizes used in the tests of the Delaunay algorithm.

- Removal. This test executed the complete clearing of collections of 10000 and 100000 objects. It removed one element at time. In order to evaluate the performance in the worst case, the object at the head of the collection was chosen to be deleted during each iteration.

- Sorting. This test invoked the sorting function on collections of 100 and 1000 points generated randomly. These size values, which are lower with respect to the tests of the other operations, have been chosen because in the Delaunay algorithm the sorting is always executed on little collections (see more details below).

Each test was executed 50 times and then the mean time was computed. They were executed on a 3.2 GHz Intel Pentium 4 processor with 1GB of RAM under Ubuntu 10.4 (OpenJDK Runtime Environment v. 1.6.0_20 and GCC v. 4.3.3). Results are reported in table 8.2 where times are expressed in milliseconds and regard the execution of all the n operations. *ArrayList* is faster than *Vec3Array* during the *insert* and the *remove* operations, whereas

it takes much time to compute the *sorting* because of the used algorithm. Indeed the method for sorting Java collections uses a modified *merge-sort* algorithm [96], which offers guaranteed $O(n \cdot log(n))$ performance. The sorting algorithm provided by the C++ STL library instead uses the *introsort* algorithm whose worst case complexity is $O(n \cdot log(n))$.

	Insert		Remove		Sort	
N. of elements	10000	100000	10000	100000	100	1000
Java	1.30	6.33	46.67	5168.21	0.13	0.42
C++	1.51	11.27	275.82	27799	0.02	0.40
Java vs C++	0.86	0.56	0.17	0.19	6.5	1.05

Table 8.2: Times report - Collection comparison

The study also analyzed the time required for the evaluation of logical conditions. Four tests were executed, taking into account the following logical conditions:

- Simple logical proposition ($var == true$)
- Disequation ($a < b$). (This is the most evaluated condition in the case study, see eq. 8.1)
- Logical disjunction of two disequations ($(a < b)||(a > c)$)
- Logical conjunction of two disequations ($(a > b)\&\&(a < c)$)

Each evaluation was executed 10000 times and each test was repeated 50 times. Table 8.3 reports average times of the tests in milliseconds. A boolean variable, initialized false and changed each execution ($var = !var$), was used in the first test and float variables (initialized with a constant values) in the others. As described in the table, Java is always faster than C++, except for what regards the evaluation of simple logical proposition.

8.2.1 The implementation details

The two implementations compute the triangulation according to the same steps, which are described in the following list.

	Prop.	Diseq.	Disj.	Conj.
N. of elements	10000	10000	10000	10000
Java	0.187	0.084	0.103	0.093
C++	0.039	0.262	0.452	0.290
Java vs C++	4.79	0.32	0.23	0.32

Table 8.3: Times report - Logical conditions evaluation comparison

1. *Initialization.* The Initialization step consists of the setting up and the sorting of the input point cloud according to their coordinates. Then four new points are inserted in order to surround the plain point cloud. These four points are used to build two main triangles (super-triangles), such that the plain point cloud lies inside their area. These triangles are stored in a collection, which is called *trianglesList*. The collection data structure was chosen accordingly to the operations that occur more often, indeed the *trianglesList* is subject to several iterations, insertions and removal. As shown in [95], *ArrayList* is the list of all the available lists in the Java framework that perform insertion, iteration and random access in the fastest way. Although removing objects from *ArrayList* requires a long time, insertions and iterations occur more often than remove operations; as a consequence *ArrayList* was chosen for implementing the *trianglesList*.

2. *Iteration.* During the iteration, each point is considered and is compared to the triangles contained in the *trianglesList*. First the condition 8.1 is checked ("*point*" stays for the current point and "*tri.circ* for the circumcircle of the current triangle).

$$point.X - tri.circ.X > tri.circ.radius \tag{8.1}$$

   - If the condition is satisfied, the current triangle is removed from the *triangleList* and will not be more considered because the current point and also the following ones surely don't lie in the circumcircle of the current triangle (i.e. the triangle respects the Delaunay

condition for all the points and it is part of the final mesh). This is guaranteed by the initial ordering of the points.

- If the condition is not satisfied, it is necessary to further investigate if the current point effectively lies in the circumcircle of the current triangle. In case it is true the Delaunay condition is not respected. Therefore the edges of the triangle are added to a specific *ArrayList* (called *edgeSet*) and the current triangle is deleted. Otherwise, if the Delaunay condition is respected, the next triangle is considered. It has to be noted that the *edgeSet* collection has been implemented as an *ArrayList* because it is sorted many times during the triangulation algorithm. Hence, the usage of *Collections.sort* method and *ArrayList* is the Java solution that allows us to save time and increase performance in the best way. In the tests of this study the maximum size of the *edgeSet* collection was never greater than 100.

When the whole *trianglesList* has been scanned, new triangles are constructed from the *edgeSet* collection and added to the *triangleList* (in these tests the maximum order of magnitude of this collection size is 10000). Note that if an edge is shared between two triangles that contain a point, then the edge is not considered. The iteration proceeds until all points have been analyzed, except for the four points created during the initialization.

3. *Completion.* The four points introduced during the initialization step and triangles having vertices in common with these four points are deleted. If there are degenerate triangles (circumcircle radius equals to 0) they are eliminated too. Finally a return result is built in form of a mesh.

Since the order of magnitude of the size of the list on which the algorithm performs more insertion is 10000, whereas the one of the collection on which the algorithm performs the sorting is 100, the time gained in Java for populating the first collection is more or less equivalent to the time lost for

sorting the second one (see table 8.2). This suggests that the time spent for managing collections is more or less the same for both the Java and C++ implementations. The evaluation of logical conditions instead seems to be not important from a performance point of view. In fact the time spent for evaluating conditions on 10000 iterations is much lower than the time spent for populating collections in more or less 100 iterations.

8.2.2 The Java HotSpot compilers

The current JVM offers a technology called HotSpot Compiler [97], which works better and faster than the pure JIT compiler. Rather than compiling each method at the first execution, the HotSpot runs the program using an interpreter for a while. During this time, in order to detect the most used and critical methods, the execution is analyzed. The collected information is then used to perform more intelligent optimizations and only the critical methods are actually compiled. This technique is called "*Adaptive Optimization*". It doesn't only produce better performance but it also reduces the overall compilation time. The adaptive optimization is continuously performed so that it adapts the performance to the users' needs.

The Java Platform Standard Edition offers a JVM that comes with two compilers: the Client and the Server versions[2]. The Client compiler is the default one and it has been specially tuned to reduce the start-up time. It is designed for client environment, in particular for applications where there is not the need of continuous computation, for example a GUI. The Server compiler instead is designed for long-running server applications, where the operating speed is more important than the start-up time. This compiler offers an advanced adaptive optimizer and supports many of the optimizations offered by the C++ compilers.

Subsections 8.2.3 and 8.2.4 report the tests executed using the client compiler whereas the server compiler is used in the experiments of subsection 8.2.5.

[2]Users can specify the compiler by means of the options "-client" and "-server". The tests on the collections and logical conditions were executed with the client compiler

8.2.3 Performance analysis

The algorithm was executed on five point clouds of different sizes: a semi-sphere, a floor, the Oxford Bunny and 2 terrains. Each point cloud was processed 50 times and the execution time was measured; then the average, the standard deviation and the confidence intervals ($1 - \alpha = 0.95$) were computed. In the first experiment each triangulation corresponds to a single program invocation, hence the program was executed 50 times per point cloud.

Table 8.4 reports the results of the test executed on the same PC presented before, running Windows XP (Sun Java 6 v. 1.6.0_23 and C++ programs compiled with MinGw v. 3.82 and GCC v. 4.5.0). Mean time, standard deviation and confidence intervals (c_1 and c_2) are expressed in milliseconds. Java vs. C++ is the ratio between the average execution time.

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	70.62	1458.8	3102.2	20344	132926
	Std Dev.	7.89	12.90	52.96	403.44	986.18
	c_1	68.38	1455.1	3087.1	20229	132646
	c_2	72.86	1462.5	3117.3	20459	133206
C++	Mean	7.50	298.13	886.25	3305.9	22908
	Std Dev	7.89	25.61	63.77	144.83	227.06
	c_1	5.26	290.85	868.13	3264.8	22844
	c_2	9.74	305.40	904.37	3347.1	22973
Java vs. C++		9.42	4.89	3.50	6.15	5.80

Table 8.4: Times report - Multiple invocation - Windows - Java Client

Referring to the table 8.4, the triangulation execution time obtained with the first point cloud (sphere) is not very truthful. Indeed the Java version is 9.42 times slower than C++ one and this value doesn't fit the ratios obtained with the other point clouds. In this case the execution time is very small and so the time required for compiling the code greatly influences the result. Note that the costs required by the compiler have a fixed part, which is the same for each point clouds. Hence the smaller is the point cloud, the greater is the

influence of the compilation overhead on the execution time.

8.2.4 Single program invocation

In order to avoid the compilation overhead the measuring strategy was changed. A second experiment was set up, where 51 triangulations were measured in a single program invocation. This kind of experiment corresponds to a long run execution, where only the first invocation of the algorithm pays the compiler costs. The execution time of the first invocation was discarded and the statistics were computed on the other 50 samples. Tables 8.5 and 8.6 reports the results obtained under Windows and Ubuntu Lucid (both with the same Java and C++ versions presented before). As expected, the average execution times decrease for Java and remain more or less the same for C++.

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	15.58	1157.2	2487.2	18108	115977
	Std Dev.	0.50	18.82	20.57	80.53	399.65
	c_1	15.44	1151.9	2481.3	18085	115863
	c_2	15.72	1162.6	2493.0	18130	116091
C++	Mean	6.56	288.75	914.06	3226	23168
	Std Dev	7.79	7.89	11.92	110.36	605.89
	c_1	4.35	286.51	910.68	3195	22996
	c_2	8.78	290.99	917.45	3257	23340
Java vs. C++		2.37	4.01	2.72	5.61	5.01

Table 8.5: Times report - Single invocation - Windows - Java Client

Table 8.5 shows that under Windows, without the compiler overhead, Java performance are always better than the results reported in table 8.4, but remain worse than the results discussed in section 8.1. Indeed in the tests the execution time ratio between Java and C++ goes from 2.37 to 5.61 against the range 1.45-2.91 reported in table 8.1. One possible reason is that the triangulation process requires an intensive use of the memory and probably the C++ version leverages the possibility of store temporary objects on the stacks, which is much faster [89].

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	6.66	869.20	2111.3	13478	92301
	Std Dev.	1.48	40.83	80.94	106.06	383.57
	c_1	6.24	857.60	2088.3	13448	92191
	c_2	7.08	880.80	2134.3	13508	92410
C++	Mean	3.23	224.01	750.09	3333.4	24277
	Std Dev	0.84	2.35	7.41	62.90	437.73
	c_1	2.99	223.35	747.99	3315.5	24152
	c_2	3.47	224.68	752.20	3351.2	24401
Java vs. C++		2.06	3.88	2.81	4.04	3.80

Table 8.6: Times report - Single invocation - Linux - Java Client

Table 8.6 shows that under Ubuntu Java is more efficient than under Windows. Indeed the ratio range goes from 2.06 to 4.04. It should be noted that the tests were executed with both the OpenJDK and the Sun JDK. However this section reports only the first one because the results were almost the same.

Another consideration can be done on the relation between the point cloud sizes and the execution time ratios. The values of the performance ratio don't show a linear trending, hence it is possible to assert that for this algorithm there is no correlation between the performance ratio and the input size.

8.2.5 JVM Server option

Tables 8.7 and 8.8 report the results obtained using the Server compiler. The tests were executed on the same machine used previously with the same configurations. Of course, only the Java version was tested, hence the C++ rows report the results of the previous experiments. Despite the optimizations provided by GCC were considered as well, they were not applied because the default *release* configuration of the OSG libraries is already tuned in order to offer the best performance.

The results in table 8.7 show that under Windows the server compiler considerably reduces the average execution time of complex operations. In

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	24.96	356.26	999.02	4873.1	30320
	Std Dev.	15.45	20.16	20.40	42.80	230.28
	c_1	20.57	350.53	993.22	4861.0	30255
	c_2	29.35	361.99	1004.8	4885.3	30385
C++	Mean	6.56	288.75	914.06	3225.9	23168
Java vs. C++		3.80	1.23	1.09	1.51	1.31

Table 8.7: Times report - Single invocation - Windows - Java Server

particular the execution times decrease 3-4 times with respect to the client compiler. The first cloud represents the unique exception. Indeed, in that case the execution is too short and so most of the iterations are executed without optimization. This is the typical case in which the larger start-up time required by the server compiler is not compensated. Regarding the other point clouds, the ratio range goes from 1.09 to 1.51 and so Java is nearly equivalent to C++.

		Sphere	Floor	Bunny	Terrain 1	Terrain 2
Number of vertices		642	10000	35947	66049	263169
Java	Mean	20.40	428.5	1253.1	4778.9	29368
	Std Dev.	19.58	20.66	56.64	172.29	250.81
	c_1	14.84	422.63	1237.0	4729.9	29297
	c_2	25.96	434.37	1269.2	4827.8	29440
C++	Mean	3.23	224.01	750.09	3333.4	24277
Java vs. C++		6.32	1.91	1.67	1.43	1.21

Table 8.8: Times report - Single invocation - Linux - Java Server

The results in table 8.8 demonstrate that also under Ubuntu the server compiler significantly improves the performances. The same considerations reported above could be applied to the results obtained with the first cloud. Referring to the other clouds the ratio range goes from 1.21 to 1.91 and the average execution times are from 2 to 3 times better than the results obtained with the client compiler.

8.3 Discussion

This chapter has described a study on the evaluation of the performance of Java with respect to C++ in robotics applications. The results obtained with the Client compiler, which works better for short-running applications, have shown that Java is from 2.72 to 5.61 times slower than C++. The use of the Server compiler, which is best tuned for long-running applications, has instead demonstrated that Java is from 1.09 to 1.91 times slower. These results show that the performance of Java are now better with respect to the tests previously documented in literature and demonstrate that the use of the Server compiler for long run applications greatly reduces the execution time.

In addition to the fact that now the performances are not so different with respect to C++, it is also important to consider that Java offers a set of interesting features.

- *Portability*: Java is designed to be platform independent and so Java software is very portable. The low level data types such as integer and float are fully defined in Java specification and are not platform dependent.

- *Reusability*: Java comes by default with a lot of common libraries for several purposes. It is also easy to deploy and reuse the developed libraries without sharing source or header files or requiring a specific compiler.

- *Maintainability*: Java is designed to forbid common bugs such as dangling pointers, casting errors, out-of-bounds arrays, stack overflows, segmentation faults and uninitialized variables.

In conclusion, the results obtained with the server compiler and these important features suggest that Java can be considered a valid alternative to C++, at least for non-real-time functionalities. New experiments can be executed in order to further confirm this thesis. These tests should especially regard the communication with external devices (USB, RS-232, etc.) and the execution of multi-thread programs.

9

Decoupling computation and coordination

A component-based system is a composition of components and the way components interact with other components and with the computational environment greatly affect the flexibility of the entire system and the reusability of individual functionality.

Supporting seamless evolution of a component-based robotic system with frequently changing requirements advocates for the separation of different design concerns (Computation, Coordination, Configuration and Communication, the famous 4CS introduced by [98]) in such a way that component features affected by robot variability can be changed independently one from the others. These include the deployment of components on different and possibly networked computing platforms, the data exchange coordination and synchronization among components, the selection and composition of components providing specific functionalities and the assignment of computational resources to each component.

Focusing on Computation and Coordination, the Computation is related with the data processing algorithms required by an application and defines **how** the functionalities are realized. Coordination instead, which should be orthogonal to the computation, is more concerned with the interaction of the components [98] and defines **when** the functionalities are used. Components can typically interact in two ways: they cooperate with each other in order to achieve a common goal and at the same time they compete for using shared resources such as memory, CPU and external devices (e.g. sensors and

actuators). Cooperation and competition are forms of interactions among concurrent activities, which overlap in the time and are interleaved with one other on a single processor. Correct interleaving of concurrent activities can be reached by means of coordination algorithms.

In order to decouple as much as possible the Computation and Coordination concerns, they can be modeled by using two different software frameworks. For what regards the computation one of the software framework presented in section 2.2.4 can be adopted. Coordination instead can be managed by means of the State Machines formal methods. In this chapter the component model offered by the Service Component Architecture (SCA)[24] is used for modeling computation while the Abstract State Machines formal method (ASM)[99] is adopted for the Coordination.

SCA was already presented in this thesis. ASM is instead an operational (read: executable) formalism that provides accurate yet practical industrially viable behavioral semantics for pseudo-code on arbitrary data structures. This specification method is tunable to any desired level of abstraction, and besides ASMs comes with a rigorous mathematical foundation, it provides rigor without formal overkill. In particular a service-oriented flavor of the ASM formalism, named SCA-ASM [100], was adopted. SCA-ASM is a formal and executable modeling language. It is based on the SCA software framework, for heterogeneous service-oriented component assembly, and on the ASM formal method, which allows to model behavioral notions of service interactions, orchestrations, compensations, and the services internal behavior.

This chapter illustrates by means of a case study how the aforementioned frameworks can be used for orthogonally modeling the computation and coordination concerns during the design of component based robotics systems.

9.1 The case study

The case study proposes a simple scenario where a laser scanner offers its services to different clients, which concurrently compete for the use of this shared resource. The problem, the requirements and an abstract solution are presented in this section.

9.1.1 The problem

The scenario is defined by the following three participants, which are illustrated in figure 9.1.

- A Laser Scanner, which executes scans of the environment on demand and writes the acquired values on a data buffer. A scan is a sequence of measures executed in a single task (for example 360 values, one for each degree). It is supposed that the Laser Scanner allows its client to request a scan from an initial angle (start) to a finale one (end), by defining the start angle and the number of steps between start and end.
- A 3D Perception application, which requests the measures of the Laser Scanner in order to generate a set of meshes that describe the surface of the objects present in the environment.
- An Obstacle Avoidance application, which requests the measures of the Laser Scanner in order to detect the obstacles along the robot path.

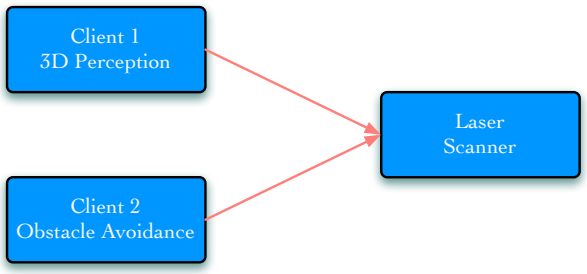

Figure 9.1: The three participants

9.1.2 The requirements

The proposed scenario is subjected to the following requirements:

1. The laser scan is an activity that requires an amount of time in order to be completed. This time is not fixed, and depends on the number of measures requested by the client. During this time the client could have

the need of executing other operations and for this reason it doesn't have to be blocked while it waits (asynchronous request of the service).

2. The clients could request a single scan or multiple scans (for example 4 scans composed each one by 20 measures).

3. While the Laser Scanner is executing a scan requested by a client A, a client B could require another scan. These requests have to be managed according to one of the following request management policies:

   - Policy 1: discard the scan request.
   - Policy 2: queue the scan request.

According to these requirements it is possible to imagine at least the following three situations, in which the Laser Scanner receives requests from its client.

The first situation is described in the sequence diagram depicted in figure 9.2. The client requests a scan to the Laser Scanner and then it waits until the end of the scan process. When the Laser Scanner finishes its work it returns to the client the measures. In this case the request is synchronous.

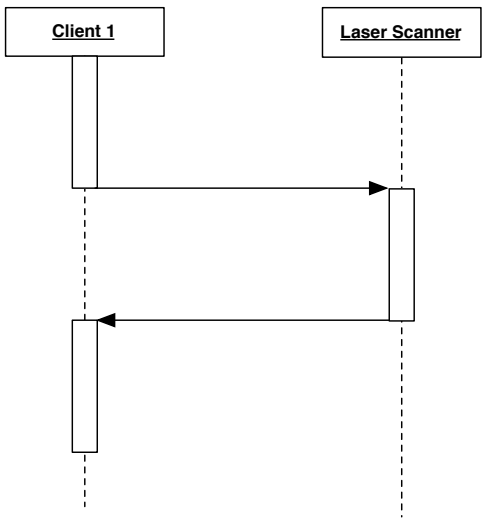

Figure 9.2: Situation 1: sequence diagram

The second situation is described in figure 9.3. The client requests a scan to the Laser Scanner and then it continues to execute its work. In this case the request has to be asynchronous.

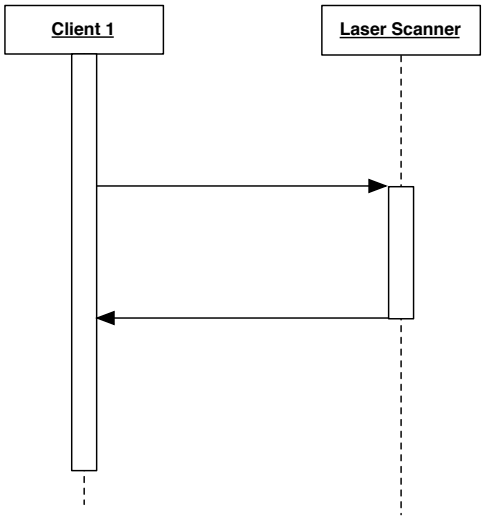

Figure 9.3: Situation 2: sequence diagram

The third situation is described in figure 9.4. In this case the client A acts as in the second situation. However while the Laser Scanner is executing the

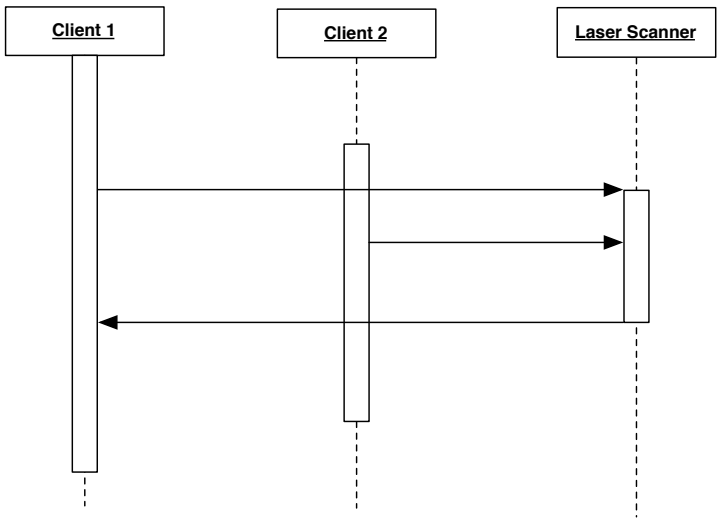

Figure 9.4: Situation 3: sequence diagram

scan requested by the client A, the client B sends another request to the Laser Scanner. This example highlights how different clients could simultaneously access the services offered by the Laser Scanner and the need of managing these different requests by means of a request management policy. In the sequence diagram of the third situation the policy 1 is applied (the third situation is not strictly related to the policy 1 but regards also the policy 2) and the service requests are asynchronous.

9.1.3 A high-level solution

The first two situations don't require a simultaneously access to the Laser Scanner services and so the client and the Laser Scanner can directly interact. Figure 9.5 illustrates how the components interact. The client A requests a scan to the Laser Scanner, which writes each measure on a Measures Buffer. Then, when the scan is finished:

- the Laser Scanner returns to the client A the measures (Situation 1); or
- the Laser Scanner notifies the client that it has completed its work. After that, the client requests the measures to the Laser Scanner (Situation 2).

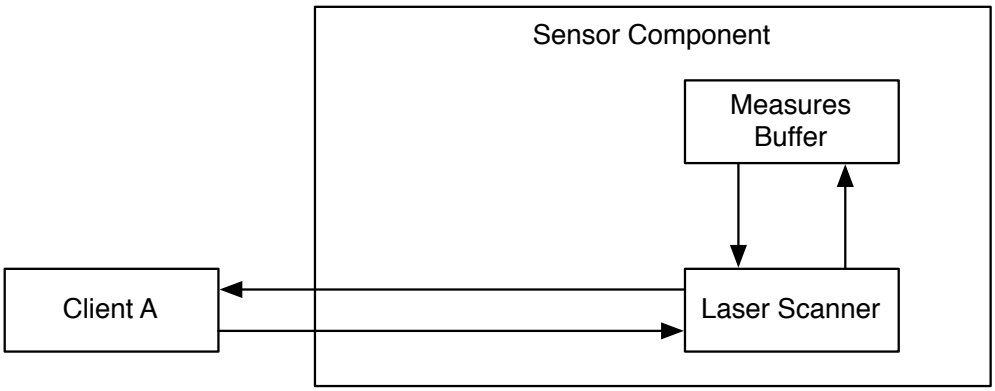

Figure 9.5: High-level solution for the situations 1 and 2

The third situation instead presents a simultaneously access to the Laser Scanner services. In this case the interactions between the clients and the

Laser Scanner have to be managed by a third part: a coordinator. The Sensor Coordinator is in charge of forwarding the client requests to the Laser Scanner and for this reason it has to manage the concurrent access of the clients. Figure 9.6 illustrates the components architecture. The interaction between components is summarized in the following list.

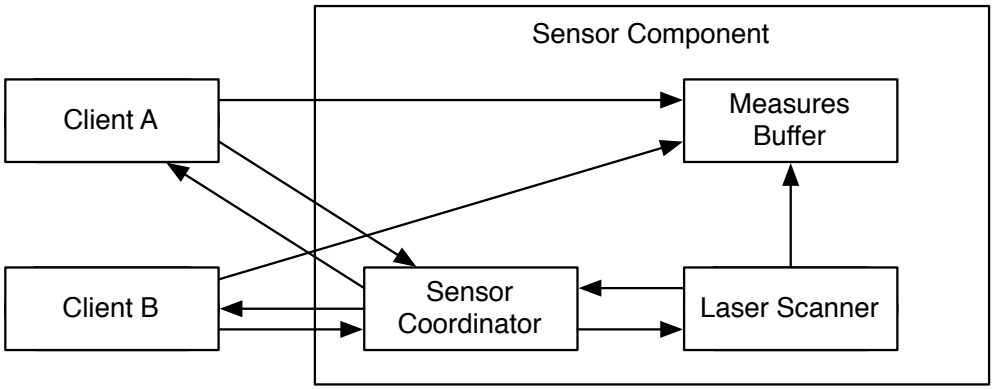

Figure 9.6: High-level solution for the situation 3

1. The Sensor Coordinator receives a request of scan from a client.

2. According to the Sensor Coordinator policy (see above) the new request could be discarded, queued or forwarded to the Laser Scanner.

3. When the request is forwarded, the Laser Scanner starts the scanning work and sends a notification to the Sensor Coordinator (*Ack*) in order to inform it that the scan has started.

4. The Laser Scanner writes each measure on the Measures Buffer until the final angle is reached.

5. The Laser Scanner sends a notification to the Sensor Coordinator (*Done*) in order to inform it that the scan is finished.

6. The Coordinator sends a notification to the client in order to inform it that the new measures are available on the Buffer.

7. The client accesses the Measures Buffer in order to read the measures.

Depending on the number of scan requested the Sensor Coordinator will forward to the Laser Scanner one or more single scans.

The Sensor Coordinator policy can be defined by means of a finite state machine. A first version is reported in figure 9.7. It implements the request management policy 1: if a request is received while the laser is already scanning the new request is discarded.

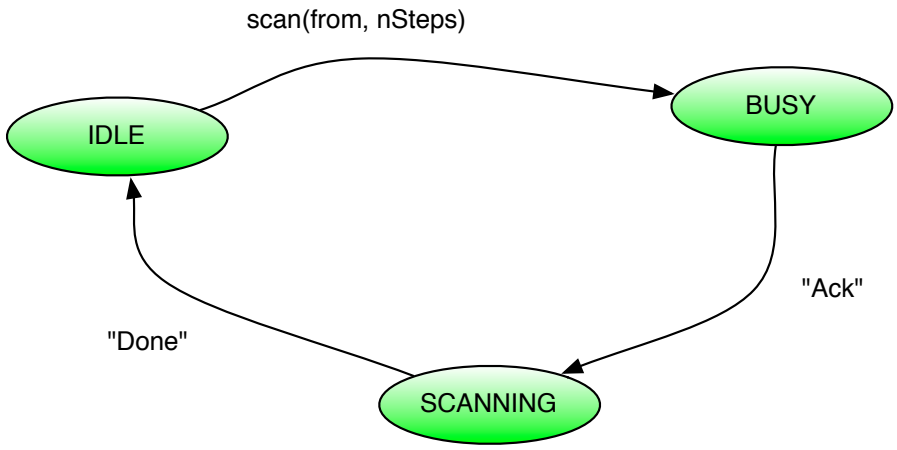

Figure 9.7: Sensor Coordinator Finite State Machine, version 1

The sates have the following meaning.

- IDLE: the Laser Scanner is idle and is ready for a new scan. In case of a scan request the Sensor Coordinator forwards it to the Laser Scanner. The Sensor Coordinator enters this state on the initialization or when the Laser Scanner returns a "*Done*" notification.

- BUSY: the laser scanner is executing the operations needed for starting the scan. The Sensor Coordinator enters this state when it has sent a scan request to the Laser Scanner. If a new scan request is received the sensor coordinator discards it.

- SCANNING: the laser is scanning and writing the measures on the Measures Buffer. The Sensor Coordinator enters this state when the

scan request is sent to the Laser Scanner and it has returned the "*Ack*" notification. If a new scan request is received the sensor coordinator discards it.

The finite state machine presented above allows a client to request a single scan and can be refined in order to support multiple scan requests. In this way the client will send a single message to the Sensor Coordinator, asking it n scans. In turn, the Sensor Coordinator will forward n single scan requests to the Laser Scanner.

Figure 9.8 illustrates the new finite state machine. It has to be noted that the Sensor Coordinator policy has been changed without modifying the functionality offered by the Laser Scanner. Furthermore this version, as well as the first, is able to satisfy single scan requests.

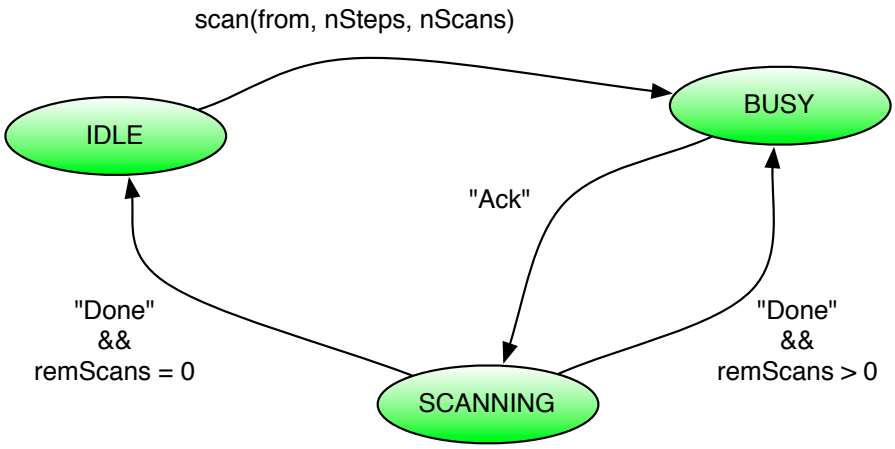

Figure 9.8: Sensor Coordinator Finite State Machine, version 2

The sates have the same meaning described above. What change are the transition rules:

- IDLE → BUSY: the transition is triggered when the Sensor Coordinator receives a scan request.
- BUSY → SCANNING: the transition is triggered when the Laser Scanner sends an "*Ack*" notification to the Sensor Coordinator.

- SCANNING → BUSY: the transition is triggered when the Laser Scanner sends a "*Done*" notification to the Sensor Coordinator and the number of remaining scans ("*remScans*") is greater than 0. In this case the Sensor Coordinator forward a new Scan request to the Laser Scanner.

- SCANNING → IDLE: the transition is triggered when the Laser Scanner sends a "*Done*" notification to the Sensor Coordinator and there are not remaining scans to execute.

Figure 9.9 shows the finite state machine that implements the policy 2: if a request is received while the laser is already scanning the new request will be queued. Also in this case the states and the functionalities provided by the laser scanner are the same. What change are the transition rules.

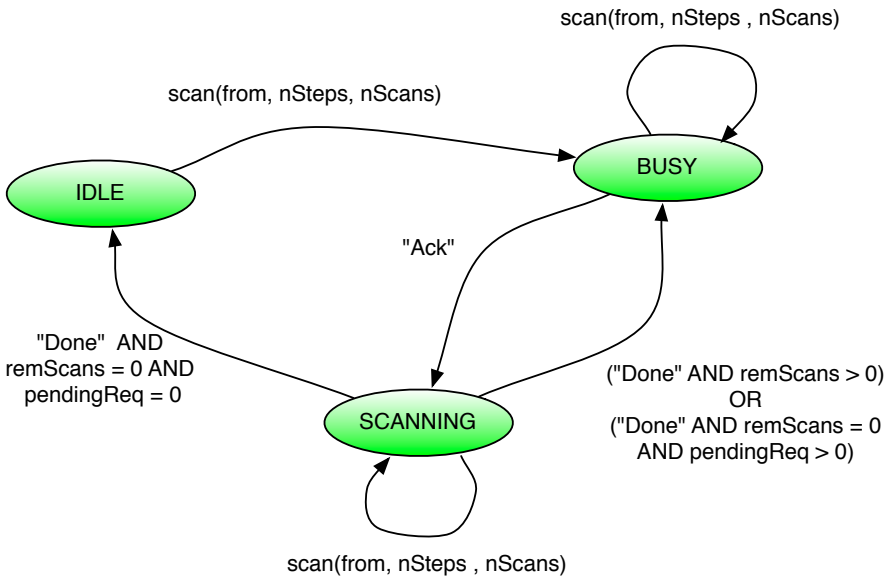

Figure 9.9: Sensor Coordinator Finite State Machine, version 3

The new transitions are described in the following list.

- IDLE → BUSY: the transition is triggered when the Sensor Coordinator receives a scan request.

- BUSY → BUSY: the transition is triggered when the Sensor Coordinator receives a scan request. The request is queued.
- BUSY → SCANNING: the transition is triggered when the Laser Scanner sends an "*Ack*" notification to the Sensor Coordinator.
- SCANNING → BUSY: the transition is triggered when
  - the Laser Scanner sends a "*Done*" notification to the Sensor Coordinator and the number of remaining scans is greater than 0, or
  - the Laser Scanner sends a "*Done*" notification to the Sensor Coordinator, there are not remaining scans to execute and the number of pending requests in queue is greater then 0.
- SCANNING → SCANNING: the transition is triggered when the Sensor Coordinator receives a scan request. The request is queued.
- SCANNING → IDLE: the transition is triggered when the Laser Scanner sends an "*Done*" notification to the Sensor Coordinator, there are not remaining scans to execute and there are not pending requests in queue.

The next section will illustrate how these finite state machines have been modeled through the ASM language.

9.2 The specification of the State Machines

The Sensor Coordinator was implemented using the SCA-ASM formalism. In the following an ASM-based (abstract) implementation of the Sensor Coordinator is reported.

9.2.1 The Abstract State Machine in a nutshell

Abstract State Machines (ASMs) are an extension of Finite State Machines (FSMs) [99] where unstructured control states are replaced by states comprising arbitrary complex data. The states of an ASM are multi-sorted first-order

structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them. The transition relation is specified by rules describing how functions change from one state to the next. Basically, a transition rule has the form of guarded update "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) :=$ t that are simultaneously executed when Condition is true.

There is a limited but powerful set of rule constructors, reported in Table 9.1, that allow to express simultaneous parallel actions (`par`) of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*, by sequential actions (`seq`), iterations (`iterate`, `while`, `recwhile`), and sub-machine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`). Furthermore, the ASM method supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synch-/Asynch Multi-agent ASMs*. In this last model, the predefined variable *self* is interpreted by each agent as itself.

Skip rule	**skip**	do nothing
Update rule	$f(t_1, \ldots, t_n) := t$	update the value of f at $t_1, \ldots, t_n$ to t
Block rule	**par** $R_1 \ldots R_n$ **endpar**	rules $R_1 \ldots R_n$ are executed in parallel
Seq rule	**seq** $R_1 \ldots R_n$ **endseq**	rules $R_1 \ldots R_n$ are executed in sequence without exposing intermediate updates
Conditional rule	**it** ϕ **then** R_1 **else** R_2 **endif**	if ϕ is true, then execute rule R_1 , otherwise R_2 fires
Iterate rule	**while** ϕ **do** R	execute rule R until ϕ is true
Forall rule	**forall** x **with** ϕ **do** R	execute R in parallel for each x satisfying ϕ
Choose rule	**choose** x **with** ϕ **do** $R(x)$	choose an x satisfying ϕ and then execute R
Macro call rule	$R[x_1, \ldots, x_n]$	call rule R with parameters $x_1, \ldots, x_n$
Let rule	**let** $x = t$ **in** R	assign the value of t to x and then execute R

Table 9.1: ASM rule constructors

Based on [99], an ASM can be defined as the tuple: (*header*, *body*, *main rule*, *initialization*).

The *header* contains the *name* of the ASM and its *signature*[1], namely all domain, function and predicate declarations.

Function are classified as *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* (read and write by an agent and by the environment or by another agent) and *output* (only write) functions.

The *body* of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. The body may also contain definitions of *invariants* to assume over domains and functions of the ASM.

The (unique) *main rule* is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body). The main rule is *closed* (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on variable assignment, but on the machine state.

The *initialization* of an ASM is a characterization of the initial states. An initial state defines initial values for domains and functions declared in the ASM signature.

Executing an ASM means executing its main rule starting from a specified initial state. A computation of an ASM M is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of M, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the transition rules which are enabled in S_n.

A lightweight notion of module is also supported. An *ASM module* is an ASM (*header*, *body*) without a main rule, without a characterization of the set of initial states, and the body may have no rule declarations. A module is

[1] *Import* and *export* clauses can be also specified for modularization.

written as an ASM with the keyword `asm` replaced by the keyword `module`.

An open framework, the ASMETA tool set [101], based on the Eclipse/EMF modeling platform and developed around the ASM Metamodel, is also available for editing, exchanging, simulating, testing, and model checking models.

9.2.2 The ASM-SCA formalism

In addition to the ASM rule constructors, other commands capturing service behavioral aspects have been provided (see [100] for more details) and formalized in terms of ASMs concepts as further actions offered by the SCA-ASM language, including constructs to express the control flow of component tasks, as well as primitive for services orchestration and interaction. Some of these actions correspond to predefined ASM rules whose AsmetaL implementation is provided in terms of an external library, named CommonBehavior, to be imported as part of a SCA-ASM module. In particular, external services are invoked in a synchronous and asynchronous manner through the following interaction (or communication) primitives:

- *wsend[lnk,R,snd]*: Sends data "*snd*" without blocking to the partner link "*lnk*" in reference to the service operation "*R*" (no acknowledgment is expected).

- *wreceive[lnk,R,rcv]*: Receives data in the location "*rcv*" from the partner link "*lnk*" in reference to the service operation "*R*; it blocks until data are received. No acknowledgment is expected.

- *wsendreceive[lnk,R,snd,rcv]*: In reference to the service operation "*R*, some data "*snd*" are sent to the partner link "*lnk*", then the action waits for data to be sent back, which are stored in the receive location "*rcv*"; no acknowledgment is expected for send and receive.

- *wreplay[lnk,R,snd]*: Returns some data "*snd*" to the partner link "*lnk*", as response of a previous "*R*" request received from the same partner link; no acknowledgment is expected.

These communication primitives rely on a dynamic domain Message that represents message instances managed by an abstract message passing mechanism, abstracting, therefore, from the SCA notion of binding. It is assumed that components communicate over links according to the semantics of the communication commands reported above and a message encapsulates information about the partner link and the referenced service name and data transferred. A data binding mechanism also guarantees a matching between ASM data types and Java data types, including structured data.

9.2.3 The Sensor Coordinator ASM - Policy 1

The following listings report the ASM implementation of the Sensor Coordinator FSM depicted in figure 9.8 (request management policy 1). To this purpose, the AsmetaL textual notation is used to write ASM models within the ASMETA tool-set. Two grammatical conventions must be recalled: a variable identifier starts with an initial $; a rule identifier begins with "*r_*".

Listing 9.1 shows the first rows of the ASM implementation. The import clauses include the ASM modules of the provided service interfaces (*SensorCoordinating* and *EventObserving*) and required interfaces (the LaserScanning interface) of the component, annotated, respectively, with "*@Provided*" and "*@Required*". The "*@MainService*" annotation when importing the SensorCoordinating interface denotes the main service (read: main component agent) that is responsible for initializing the component state (in the predefined "*r_init*" rule) and, eventually, for the start-up of the other agents by assigning programs to them. The signature of the machine contains declarations for:

- References (shared functions annotated with *@Reference*), which are abstract access endpoints to services.
- Back references to requester agents (shared functions annotated with *@Backref*).
- Declarations of the following ASM domains and functions, which are used by the component for internal computation only.

- Enumeration domain "*State*" defines the possible control states of the ASM shown in subsection 9.1.3.
- Variable "*ctl_state*" stores the current control state.
- Variable "*paramScan*" stores the input parameters of the "*request*" function.
- Variable "*from*" and "*steps*" store the start position and the number of measures that compose a scan request received from a client.
- Variable "*remScans*" stores the number of remaining scans to do.
- Variable "*event*" stores the input parameter of the "*update*" function.

```
1  module SensorCoordinator
2  import STDL/StandardLibrary
3  import STDL/CommonBehavior
4
5  //@MainService
6  import SensorCoordinating
7  //@Provided
8  import EventObserving
9  //@Required
10 import LaserScanning
11 export *
12
13 signature:
14 //@Reference
15 shared laserScanning : Agent -> LaserScanning
16 //@Backref
17 shared clientSensorCoordinating : Agent -> Agent
18 //@Backref
19 shared clientEventObserving : Agent -> Agent
20
21 enum domain State = {IDLE | BUSY | SCANNING}
22 //Internal properties
23 controlled ctl_state : Agent -> State
24 controlled paramScan : Agent -> Prod(Real,Integer,Integer)
25 controlled from : Agent -> Real
26 controlled steps : Agent -> Integer
27 controlled remScans : Agent -> Integer
28 controlled event : Agent -> String
```

Listing 9.1: The ASM implementation header

The body of the ASM, which starts with the keyword "*definitions:*", includes definitions of services (ASM transition rules annotated with *@Service*) "*r_request*" and "*r_update*", the definition of the main transition rule "*r_SensorCoordinator*" (that takes by convention the same name of the component module) and the transition rule with the predefined name "*r_init*", which is in turn invoked in the initialization rule of the container composite to initialize the internal state (controlled functions). Another utility rule, named "*r_acceptRequest*", has been introduced for modularization purposes and to advance the control state of the machine according to the arriving service requests properly.

Listing 9.2 reports the body of the "*r_request*" rule, which is in charge of requesting a scan to the laser scanner.

```
1  definitions: //definitions of named ASM transition rules
2  //@Service
3  rule r_request($a in Agent,$from in Real,$steps in Integer, $nScans in Integer)=
4    par
5      ctl_state($a) := BUSY
6      from($a) := $from
7      steps($a) := $steps
8      remScans($a) := $nScans − 1
9      r_wsend[laserScanning($a),"r_scan(Agent,Real,Integer)",($from,$steps)]
10   endpar
```

Listing 9.2: The *"r_request"* rule

When this rule is called, it executes the following operations in a parallel way:

1. Sets the state of the ASM to *BUSY*.
2. Stores the parameters of the requested scan in the variables "*from*", "*steps*" and "*remScans*".
3. Calls the function "*scan*", which is provided by the service Laser Scanning.

Listing 9.3 reports the body of the "*r_update*" rule, which is in charge of receiving the notification from the laser scanner and updating the control state of the ASM.

```
1  //@Service
2  rule r_update($a in Agent, $event in String) =
3    if (ctl_state($a)=BUSY and $event="Ack")
4      then ctl_state($a) := SCANNING
5    else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)>0)
6      //continue with next scan
7      then par
8        ctl_state($a) := BUSY
9        remScans($a) := remScans($a)-1
10       r_wsend[laserScanning($a),"r_scan(Agent,Real,Integer)",(from($a),steps($a))]
11     endpar
12   else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)=0)
13     then ctl_state($a) := IDLE
14   endif endif endif
```

Listing 9.3: The *"r_update"* rule

When this rule is called, it executes the following operations:

1. If the current control state is *BUSY* and the notification is an "*Ack*": the rule sets the control state to *SCANNING*.

2. If the current control state is *SCANNING* and the notification is a "*Done*" and the number of remaining scans is greater than 0, the rule executes the following operations in a parallel way:

   - Sets the control state to *BUSY*.
   - Decrements the number of remaining scans.
   - Calls the function "*scan*", which is provided by the service Laser Scanning.

3. If the current control state is *SCANNING* and the notification is a "*Done*" and there are not remaining scans to do, the rule sets the control state to *IDLE*.

Listing 9.4 reports the body of the "*r_acceptRequest*" rule. It is in charge of processing the request received from the clients of the coordinator and the Laser Scanner. For this reason it receives as input a string, which contains the request itself. When this rule is called, it sequentially executes the following operations:

1. If the client has requested a new scan(*r_request*):
   (a) It removes the request from the requests stack (operation "*r_wreceive*") and stores the input parameters in the variable "*paramScan*".
   (b) If the parameters are defined (condition "*isDef*") the rule calls the rule "*r_request*" (see above)
2. If the Laser Scanner has sent a notification (*r_update*):
   (a) It removes the request from the requests stack (operation "*r_wreceive*") and stores the input parameter in the variable "*event*".
   (b) If the parameter is defined the rule calls the rule "*r_update*" (see above).

```
1  rule r_acceptRequest ($a in Agent, $r in String) =
2    if (ctl_state($a)=IDLE and $r="r_request(Agent,Real,Integer,Integer)")
3      then seq
4        //first scan
5        r_wreceive[clientSensorCoordinating($a),"r_request(Agent,Real,Integer,Integer)",paramScan($a)
             ]
6        if (isDef(paramScan($a)))
7          then
8            r_request[$a,first(paramScan($a)),second(paramScan($a)),third(paramScan($a))]
9        endif
10     endseq
11   else if (not ctl_state($a)=IDLE and $r="r_update(Agent,String)")
12     then seq
13       r_wreceive[clientEventObserving($a),"r_update(Agent,String)",event($a)]
14       if (isDef(event($a)))
15         then r_update[self,event($a)]
16       endif
17     endseq
18   endif endif
```

Listing 9.4: The *"r_acceptRequest"* rule

The rule is implemented in such a way that all the scan requests received while the scanner is already scanning are discarded (that's what the policy 1 defines).

Listing 9.5 reports the body of the "*r_SensorCoordinator*" rule. It is the main rule of the agent and is called every times a client requests a service

offered by the Sensor Coordinator. This rule simply forwards the request to the "*r_acceptRequest*" rule (see above).

```
1 //Main agent program
2 rule r_SensorCoordinator =
3   let($r = nextRequest(self)) //Select the next request(if any)
4   in if isDef($r)
5       then r_acceptRequest[self,$r] //Handle the request $r
6     endif
7   endlet
```

Listing 9.5: The *"r_SensorCoordinator"* rule

Listing 9.6 reports the body of the "*r_init*" rule. It is called in order to initialize the agent. This rule simply sets the status of the agent to *READY*, the control state to *IDLE* and initializes the scan parameters to 0.

```
1 //Rule invoked for the startup of the component main agent
2 rule r_init($a in SensorCoordinating) = //to initialize the component state
3   par
4     status($a) := READY
5     ctl_state($a) := IDLE
6     from($a) := 0.0
7     steps($a) := 0
8     remScans($a) := 0
9   endpar
```

Listing 9.6: The *"r_init"* rule

9.2.4 The Sensor Coordinator ASM - Policy 2

This subsection illustrates how the abstract state machine presented above can be modified in order to implement the request management policy 2 (figure 9.9).

Listing 9.7 shows the new implementation of the rule "*r_acceptRequest*". The rows 1-15 are the same of the previous implementation. A third "*if*" condition was added (rows 16-20) in order to manage the requests received while the control state is different from *IDLE*. In fact, when the control state is not *IDLE* and the request received is a scan request, the rule sequentially

executes the following operation:

```
1  rule r_acceptRequest ($r in String) =
2    if (ctl_state(self)=IDLE and $r="r_request(Agent,Real,Integer,Integer)")
3      then seq //first scan
4        r_wreceive[clientSensorCoordinating(self),"r_request(Agent,Real,Integer,Integer)",paramScan(
             self)]
5        if (isDef(paramScan(self))) //direct service invocation
6          then r_request[self,first(paramScan(self)),second(paramScan(self)),third(paramScan(self))
               ]
7        endif
8      endseq
9    else if (not ctl_state(self)=IDLE and $r="r_update(Agent,String)")
10     then seq
11       r_wreceive[clientSensorCoordinating(self),"r_update(Agent,String)",event(self)]
12       if (isDef(event(self)))
13         then r_update[self,event(self)]
14       endif
15     endseq
16   else if (not ctl_state($a)=IDLE and $r="r_request(Agent,Real,Integer,Integer)")
17     then seq //first scan
18       r_wreceive[clientSensorCoordinating($a),"r_request(Agent,Real,Integer,Integer)",paramScan($a)
             ]
19       append(pendingRequests($a), (first(paramScan($a)),second(paramScan($a)),third(paramScan($a)
             ))
20     endseq
21   endif endif endif
```

Listing 9.7: The *"r_acceptRequest"* rule - Policy 2

- It removes the request from the requests stack (operation "*r_wreceive*") and stores the input parameters in the variable "*paramScan*".
- It puts the parameters in a queue called "*pendingRequests*" (operation "*append*"). This queue is defined in the header of the state machine and is the equivalent of an array, which stores triplets (a real and two integers: the three parameters of a scan request).

All what is needed to queuing the scan requests is defined in this rule. Indeed, as mentioned above, this is the rule that implements the request management policy.

The changes in the rule "*r_update*" are instead necessary in order to manage the requests queue. They are reported in the Listing 9.8. The rows

1-10 are the same of the previous implementation. The third "*if*" condition was modified and split in two new conditions by adding a control on the number of pending requests. The following list describes the rows 11-20.

```
1  rule r_update($a in Agent, $event in String) =
2    if (ctl_state($a)=BUSY and $event="Ack")
3      then ctl_state($a) := SCANNING
4    else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)>0)
5      //continue with next scan
6      then par
7        ctl_state($a) := BUSY
8        remScans($a) := remScans($a)-1
9        r_wsend[laserScanning($a),"r_scan(Agent,Real,Integer)",(from($a),steps($a))]
10     endpar
11   else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)=0 and lenght(
         pendingRequests($a))>0)
12     then let($tmp=first(pendingRequests($a))) in
13       seq
14         pendingRequests($a) := excluding(pendingRequests($a),$tmp)
15         r_request[($a),first($tmp),second($tmp),third($tmp)]
16       endseq
17     endlet
18   else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)=0 and length(
         pendingRequests($a))=0)
19     then ctl_state($a) := IDLE
20   endif endif endif endif
```

Listing 9.8: The *"r_update"* rule - Policy 2

1. If the current control state is *SCANNING* and the notification is a "*Done*" and there are not remaining scans to do and the number of pending requests is greater than 0: the rule sequentially

   (a) removes the first request from the queue,

   (b) calls the rule "*r_request*" by passing the parameters retrieved from the queue.

2. If the current control state is *SCANNING* and the notification is a "*Done*" and there are not remaining scans to do and there are not pending requests: the rules sets the control state to *IDLE*

9.3 The case study implementation

The scenario presented above was implemented in SCA and ASM. Figure 9.10 illustrates the SCA Sensor Composite, which represents the composite component defined in figure 9.6. The clients are not present in this diagram, but they can interact with the Sensor Coordinator and with the Measures Buffer through the services promoted by the composite. In particular a client could request a scan by means of the service "*Sensor Coordinating*" and could access the Measures Buffer by means of the service "Measures Buffer Reading".

- Component 1: Measures Buffer
  - Measure Buffer Writing: it is used for writing a measure on the buffer (provided).
  - Measure Buffer Reading: it is used for reading a measure from the buffer (provided).

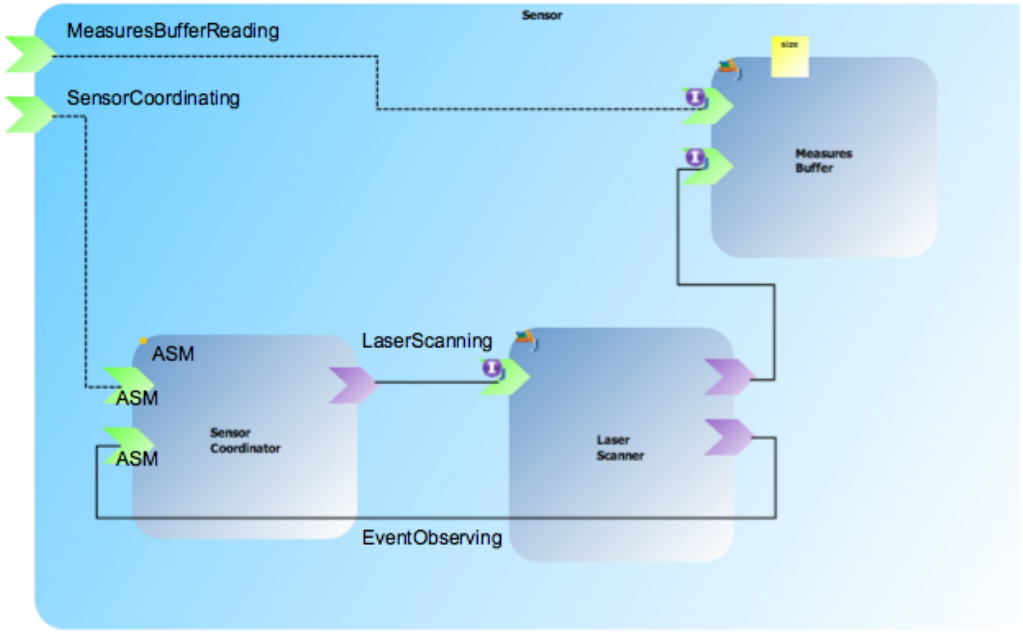

Figure 9.10: The Sensor Composite

- Component 2: Laser Scanner
  - Laser Scanning: it is used for starting a scan. The scan operation provided by the service requires two parameters: *from* and *numOfSteps* (provided).
  - Measure Buffer Writing (required).
- Component 3: Sensor Coordinator
  - Sensor Coordinating: it is used in order to request a number of scans. The scan operation provided by the service requires three parameters: *from, numOfSteps, and numOfScans* (provided).
  - Event Observing: it is used in order to notify the coordinator when the scan process starts and when it finishes (provided).
  - Laser Scanning (required).

9.3.1 The interfaces and the data structures

The interfaces of the services and the used data structures are reported in the listing 9.9. To be noticed that the interfaces of the Sensor Coordinator are defined both in Java and ASM. Indeed, despite the Sensor Coordinator is implemented with ASM, its interfaces have been defined also in Java because in this way the Java interpreter can recognize them and in this way it is possible to can call the Sensor Coordinator services from a component implemented in Java without syntax errors.

In order to manage the notification received from the Laser Scanner the Sensor Coordinator also implements the interface reported in the listing 9.10. So far it is used as a Service in order to simulate a callback, because the callbacks are not yet supported in the SCA-ASM Eclipse plugin. It will be used as a SCA Callback as soon as this further feature will be implemented and supported.

The ASM definitions of the sensor coordinator provided interfaces are reported in the listing 9.11 using the AsmetaL notation. They are ASM modules containing only declarations of business agent types, declared in

```
1  public interface MeasuresBufferReading {
2
3      public LaserScan getScan();
4  }
5
6  public interface MeasuresBufferWriting {
7
8      public void writeMeasure(LaserMeasure measure);
9  }
10
11 public interface LaserScanning {
12
13     /**
14      * @param from: point from which the laser starts the scan
15      * @param numOfSteps: number of steps of the scan */
16     @OneWay
17     public void scan(double from, int numOfSteps);
18 }
19
20 public interface SensorCoordinating {
21
22     /**
23      * @param from: point from which the laser starts the scan
24      * @param numOfSteps: number of steps of the scan
25      * @param numOfScans: number of scans required */
26     @OneWay
27     public void request(double from, int numOfSteps, int numOfScans);
28 }
```

Listing 9.9: The Java interfaces of the components

```
1  public interface EventObserving {
2
3      /**
4       * @param event: it describe the type of event.
5       * For the laser scanner valid values are "Ack" and "Done"
6       */
7      public void update(String event);
8  }
```

Listing 9.10: The Java EventObserving interface

```
1  //@Remotable
2  module SensorCoordinating
3  import STDL/StandardLibrary
4  import STDL/CommonBehavior
5  export *
6
7  signature:
8  // the domain defines the type of this agent
9  domain SensorCoordinating subsetof Agent
10 // out is a function that implements the provided service
11 out request: Prod(Agent,Real,Integer,Integer) -> Rule
12 definitions:
13
14
15 //@Remotable
16 module EventObserving
17 import STDL/StandardLibrary
18 import STDL/CommonBehavior
19 export *
20
21 signature:
22 domain EventObserving subsetof Agent
23 out update: Prod(Agent,String) -> Rule
24 definitions:
```

Listing 9.11: ASM definition of the Sensor Coordinating interface

terms of subdomains of the predefined ASM Agent domain, and of business functions, declared as parameterized ASM out functions.

9.4 Discussion

This chapter has presented, by means of a case study, an approach regarding the coordination of functionalities in the context of a robotics application. The approach is based on SCA and ASM and has the goal of demonstrating how the Computation and the Coordination concerns can be decoupled by separately modeling them.

The use of two different frameworks for modeling the two concerns (SCA for computation and ASM for coordination) improves the level of flexibility and reusability. Thanks to this orthogonal separation, it was possible to implement two different coordination policies without modifying the implementation of the services offered by the Laser Scanner. In the same way it is possible to modify the implementation of the Laser Scanner services without modifying

the coordination state machine.

Moreover ASM allows automatic validation and verification of the correctness and reliability of single components taken in isolation. It also consents runtime monitoring of the components and self-adaptation.

These positive features, the results demonstrated through the case study, the level of maturity of these frameworks and their significant spread in other domains such as the web services, support and advocate the thesis according to which the integration of SCA and ASM promises good results also in the robotics field.

10

Differential Constraints Modeling Language

Differential equations are widely used for modeling kinematics and dynamics constraints of mobile robots, for example in simulation and sampling-based path planning algorithms. Differential models express relations between configuration variables. The possible states of a mobile robot are represented in the state space X and each state represents a particular configuration of the robot. For wheeled mobile robots each state $\vec{x} \in X$ is $\vec{x} = (x, y, \theta)$, where x and y represent the position of the robot in the plane and θ is its orientation. In the same way it is possible to define the action space U, which is the set of all the possible actions on all the possible states (an action is a response of the robot, which changes its current state, to an external input). Thus a differential model can be represented as $\dot{\vec{x}} = f(\vec{x}, \vec{u})$ where $\vec{x} \in X$ is the starting state, $\vec{u} \in U$ is the action applied to the model and f is a function, called state transition function, which defines the relation between state space and action space [102, Ch. 13]. The results are expressed in terms of velocities $\dot{\vec{x}}$ and the outcome of their integration represents the future states that satisfy the kinematics constrains.

These models are widely used in simulation algorithms (which, given the starting configuration and the action vector, compute the final configuration of the robot) and sampling-based motion planning algorithms (that sample collision free configurations, which need to be compatible with the differential constraints, for computing the path toward the goal). These algorithms, in order to compute final configurations, have to solve the differential equations

and this is typically done by means of solvers, which use numerical approximation techniques. However solvers require that differential equations are implemented in the source code fulfilling specific interfaces, and implementing these equations is usually error prone and not trivial. Another problem is that differential models are usually hard-coded in the implementation of these algorithms, hence the algorithm implementation is hard-coupled to the specific robot.

In order to achieve higher flexibility, modularity and easier extensibility with respect to the current situation and to solve the problems presented before, a higher level representation of those models is needed. Domain specific languages (DSLs) provide this higher level representation. DSLs are simple formal languages, usually declarative, used to represent domain specific knowledge using some sort of syntax. DSLs let you describe easily a scenario in a specific domain.

The syntax and semantic of these DSLs are designed explicitly to describe only the knowledge of a specific domain and thus DSLs provide an advantage in terms of expressiveness and ease of use compared to general purpose languages for the specific domain. They can also improve productivity and maintenance costs. Conversely DSLs are usually less expressive than general-purpose programming languages out of their domain. More details on DSLs can be found in [103] and [104].

DSLs have also other advantages over general-purpose languages while expressing knowledge in the specific domain:

- being less expressive and complex than general purpose languages, DSLs can be used also by people that are not expert programmers;
- manual implementation of the model can require experience in computer programming and it is error prone while you can usually generate code from a DSL document in an in automated way;
- the model itself can be used as documentation;
- ease the communication between programmers and domain experts;

- ease the description of the scenario;
- can decouple the representation of the model from the technologies and interfaces used in the implementation.

This chapter presents DCML (Differential Constraints Modeling Language) a Domain Specific Language that allows the description of differential models with a high level of abstraction from implementation details. Moreover DCML provides developers with a model to text transformation for generating the code that implements the differential equations starting from the differential model, which describes the relations between state space and action space of a specific robot. This implementation can be used by motion planning algorithms in order to simulate the behavior of the robot itself.

The development of DCML, as well as the development of the tools described in the chapter 4, has followed the Model Driven Engineering (MDE) approach. A goal of this research is indeed demonstrating that the MDE can be applied with good results also to the representation of differential models.

10.1 Related works

Despite the research described in this chapter focus only on the use of differential models in sampling-based motion planning algorithms, differential models are widely used also in other robotics fields. They can be used to describe several kinds of robots: [102] and [105] present differential models of some wheeled mobile robots under kinematics and dynamics constraints, while [106] shows a model of an hexapod robot. An extension of differential models, that can take into account also dynamics constraints, are phase-space models that consider also accelerations and can then be described as $\ddot{x} = f(\dot{x}, x, u)$. Each second order model can be converted in a first-order model, which is a differential model, using a *phase space*, that has more dimension than the *state space* of the second order model. In this way it is possible to represent, using differential models, also dynamic constraints. In the same way a *kth*-order model can be expressed as a differential model using an adequate *phase space*.

Thus differential models can be used for motion planning under kinematics and dynamics constraints, as shown in [102, Ch. 14].

A first way of defining differential models with a higher level of abstraction than hard-coded solutions is using Simulink[1]. It provides developers with a toolchain for defining, through block diagrams, differential models and generating from these diagrams C and C++ code that implements them. This approach is not flexible enough because it does not allow the generation of code in other programming languages and also because it does not allow developers to customize the generated code, in terms of interface and optimization.

Another approach is using a dedicated Domain Specific Language. Literature presents, up to now, a few DSLs to describe differential equations. The MyFEM language, presented in [107], is a DSL for the definition of partial-differential equations using a subset of the Python language. It allows the generation of C++ code that implements the model defined in MyFem but it has not got an IDE. Scalation [108] is an embedded DSL defined over the Scala programming language, and it has a package that allows the representation of systems of differential equations. These approaches use subsets of existing programming languages to define the DSLs. This has some advantages, such as less learning time, but it has also the big drawback that the resulting DSL is too close to the general purpose language and thus it has less abstraction than a dedicated DSL and requires too much effort to be used by programmers that are not expert. Another drawback of both approaches is that the syntax used to express differential equations is too far from the mathematical formalism because it is tied to the syntax of native programming languages.

Other approaches, such as the one in [109], which defines a specification language for partial differential equations on a union of rectangles, or [110], which defines differential equations using the arrow notation, are quite complex and difficult in order to be used as an effective aid to developers. A common disadvantage of all these approaches is that they are tied to work only with a fixed set of numerical solvers.

Given the fact that existing solutions for representing differential equations

[1]Simulink - www.mathworks.com/products/simulink/

are too complex or do not offer enough flexibility in the code generation phase, a new DSL for representing differential models has been designed and developed. DCML offers two advantages with respect to existing solutions. Firstly the syntax used to describe differential equations is close to the mathematical one, and secondly DCML is not tied to work with a fixed set of differential equations solvers.

10.2 Differential Constraints Modeling Language

DCML allows users to describe constraints that affect mobile robots by means of differential models[2]. This allows users to focus on the description of the differential equations.

A simple model, taken from [102], that can be used to describe the constraints of a differential drive (a mobile robot with two independent wheels) is presented in the system of equations 10.1.

$$\begin{aligned} \dot{x} &= \frac{r}{2}(u_l + u_r)cos\theta \\ \dot{y} &= \frac{r}{2}(u_l + u_r)sin\theta \\ \dot{\theta} &= \frac{r}{L}(u_r - u_l) \end{aligned} \tag{10.1}$$

The state vector (x, y, θ) represents the cartesian position of the robot while the action vector $u = (u_l, u_r)$ represents angular velocities of the wheels, r is the radius of each wheel while L is the distance between the two wheels.

10.2.1 The grammar

Listing 10.1 shows the DCML document representing the differential drive presented above. A document that conforms to DCML can describe several models and for each model the user can specify:

[2]The DCML Eclipse Tool and some example are available at `http://robotics.unibg.it/software/dcml/`.

- The action space (keyword **ACTION**). In the example two actions are specified: u_l and u_r.
- The dimensions of the configuration space (keyword **CONFIG**). In the example each configuration can be expressed in terms of x, y, θ.
- The parameters of the model (keyword **PARAM**). In the example r and L.
- The state transition function of the model, which can be expressed by means of differential equations in an understandable way.
- Some temporary variables, which can be used to ease the definition of differential constraints (keyword **VAR**).
- Some constant values, different from the predefined ones, such as π and e (keyword **CONST**).
- The package (or namespace) in which the source code will be created (keyword **PACKAGE**). The example specifies that the source code has to be created in the package *robotics.models*.
- If the model definition isn't expressive enough, further comments can be added with a JavaDoc style notation.

```
1  BEGIN DifferentialDrive
2   PACKAGE : robotics.models;
3   ACTION : u_l, u_r;
4   PARAM : L, r;
5   CONFIG : x, y, theta;
6
7   d(x) = r / 2 * (u_l + u_r) * cos(theta);
8   d(y) = r / 2 * (u_l + u_r) * sin(theta);
9   d(theta) = (r / L) * (u_r - u_l);
10 END;
```

Listing 10.1: Differential Drive model

While actions, configurations and differential equations are mandatory, the other elements are useful only for describing more complex models (see Section 10.3).

The grammar of DCML, which is described in Listing 10.2, is presented below. A grammar has four main components, [111]:

1. a set Σ of terminals, which are the basic symbols that form valid instructions of the developed language;
2. a set V of non-terminals, that are syntactic variables that represent set of strings;
3. a non-terminal $s \in V$ that acts as start symbol;
4. a set P of productions, that define how terminals and non-terminals can be combined in order to generate valid strings.

For the DCML grammar:

1. The set Σ is equal to { *"**"*, *"*\"*, *"BEGIN"*, *"END"*, *";"*, *"PACKAGE"*, *":"*, *"ACTION"*, *"PARAM"*, *"CONST"*, *"CONFIG"*, *"VAR"*, *","*, *"+"*, *"-"*, *"*"*, *"\"*, *"("*, *")"*, *"d("*, *ID*, *PCKG_ID*, *NUM*, *COMMENT*} where ID represents an alphanumerical identifier, NUM is a numeric literal and $PCKG\_ID$ is a package identifier.
2. The set V is composed by {*modelList, model, package, actions, params, constants, configurations, variables, varList, constList, varDef, constDef, assignments, assignment, var, expr, term, factor, paramList*}.
3. The start symbol is *modelList*.

The DCML grammar is expressed using the Extended Backus-Naur Form (EBNF) [112] that describes each production in the form $A \rightarrow f(V_1, \ldots, V_n, \alpha_1, \ldots, \alpha_m)$ where $A, V_1, \ldots, V_n \in V$, $\alpha_1, \ldots, \alpha_m \in \Sigma$ and f is a function that concatenates symbols using regular expressions. A production means that the non-terminal on the left hand side can be replaced by the regular expression on the right hand side of the $\rightarrow$ operator.

The first production (row 1) involves the *modelList* terminal, and means that a document of DCML must contain at least one model. The second production describes the syntax of each model, it must be enclosed between a *"BEGIN"* instruction and an *"END"* instruction and the ID must be unique in the document. Symbols enclosed between square brackets are optional. In the differential drive example the ID is *DifferentialDrive*.

```
1  modelList −> model(model)∗
2  model −> [``/∗∗" COMMENT ``∗/"] BEGIN ID [package] actions [params] [constants]
       configurations [variables] assignments END``;"
3  package −> PACKAGE ``:" PCKG_ID``;"
4  actions −> ACTION ``:" varList ``;"
5  params −> PARAM ``:" varList ``;"
6  constants −> CONST ``:" constList ``;"
7  configurations −> CONFIG ``:" varList ``;"
8  variables −> VAR ``:" varList ``;"
9  varList −> varDef (``," varDef)∗
10 constList −> constDef (``," constDef)∗
11 varDef −> ID
12 constDef −> ID ``=" (``+" | ``−")NUM
13 assignments −> assignment (assignment)∗
14 assignment −> var ``=" expr``;"
15 var −> ID | ``d(" ID ``)"
16 expr −> term ( (``+" | ``−")term )∗
17 term −> factor ( (``∗" | ``/")factor )∗
18 factor −> NUM | ``(" expr ``)" | var[``("paramList``)"]
19 paramList −> expr(``," expr)∗
```

Listing 10.2: DSL Grammar

The productions at rows 4,5,7,8 define that, after the specific keywords, a list of variable declarations is needed. These lists represent, respectively, actions, parameters, configurations, and temporary variables. Each variable list, represented by the non-terminal *varList*, is made up of one or more variable declarations (row 9), each one consisting in an ID, as shown in the production at row 11, that must be unique in the model.

In a similar way the production whose head consists of non-terminal *constants* (row 6) defines that, after the keyword *"CONST"*, a list of constant declarations *constList* (row 10) is needed. Each constant declaration is made up of an ID, unique in the model, and a numeric literal, as shown in the

production 12.

The non-terminal *assignments* can be replaced by a list of differential equations. Each equation is defined as $var = expr$, where var is a non-terminal that represents, as shown in production 15, either a differential variable of the first order, or an already defined identifier. An algebraic expression, represented by $expr$, is composed by predefined functions, such as sin or cos, parenthesized expressions, numeric literals, predefined constants, such as π, or instances of the var non-terminal and also the usual mathematical operators $+, -, *, /$.

10.2.2 From model to code

Once a model has been written according to the grammar described above, it can be validated and transformed in the code that implements the differential equations. The process of validating the model and generating the source code is depicted in figure 10.1.

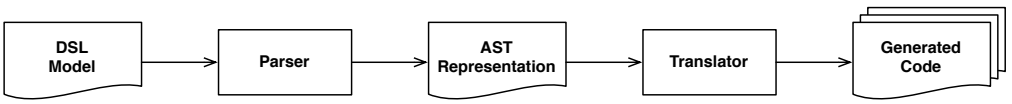

Figure 10.1: Validation and Code generation process

It can be divided in two phases. During the first one, the parsing phase, the document is validated. The parser checks that the document is correct, both from a syntactic point of view (it must respect the syntactic rules) and also from a semantic point of view (e.g. the parser checks that the document does not contains undeclared variables, non unique identifiers or function invocations with a wrong number of parameters). In this phase the parser builds, starting from the document, the Abstract Syntax Tree (AST) that is an intermediate representation of the model. The AST is a tree representation of the syntactic structure of the model enriched with some useful semantic information elaborated during the parsing phase. Figure 10.2 shows a part of the AST that is created starting form the first differential equation of the differential drive example.

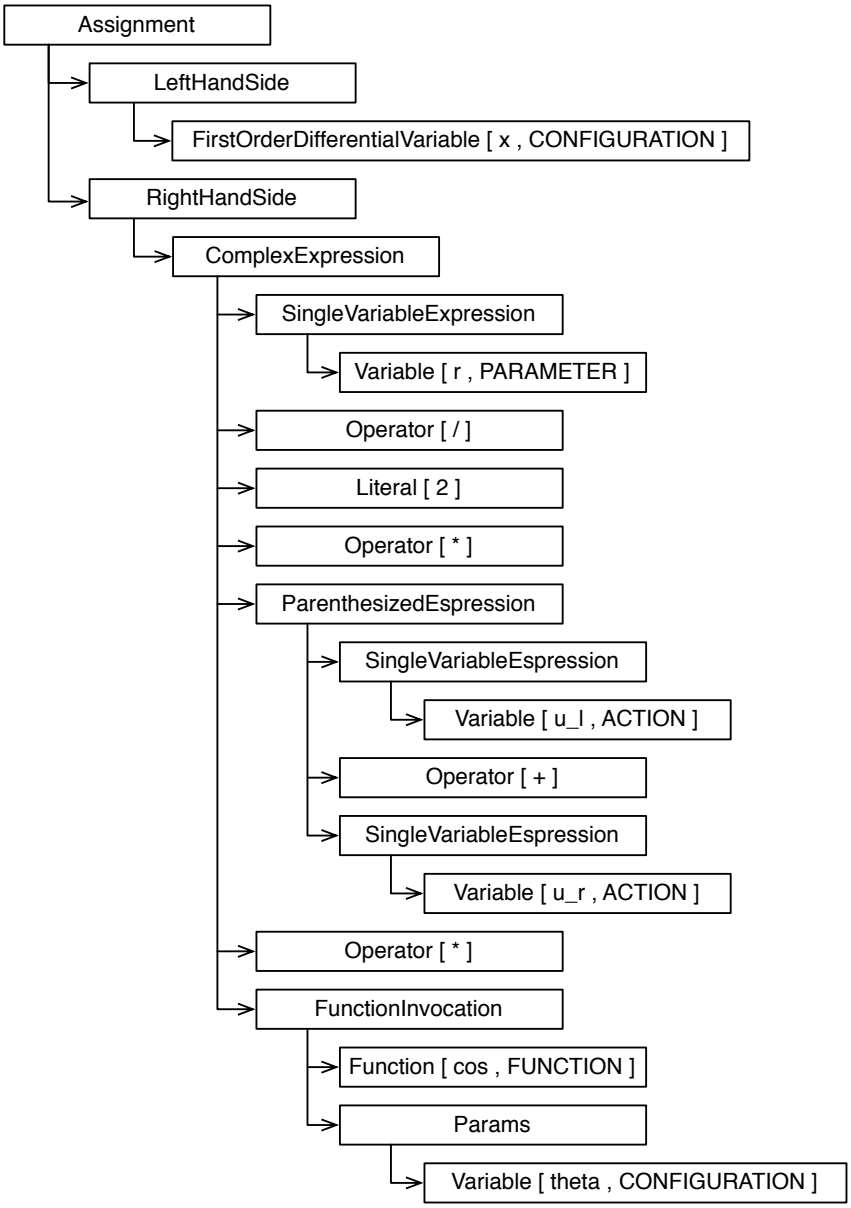

Figure 10.2: A snippet of an AST that describes a differential equation

Taking the AST as input, the second phase (i.e. the translation phase) creates the code that implements the differential model by means of a general purpose programming language. This can be done by simply visiting the AST, because it is a tree structure bearing all the information needed for the translation.

The decision of generating an intermediate representation by means of ASTs, instead of performing directly the translation during the parsing phase, has some advantages:

- allows the validation of the model without performing the translation;
- by decoupling the translation form the parsing phase it is possible to develop and use several translators, which target several programming languages and/or numerical solvers, without modifying the parser. This is possible because the parsing phase is completely separated by details regarding the generation of the code.

Despite this solution is a bit less efficient than performing the translation during the parsing phase, it has great advantages in terms of extensibility and flexibility.

The Java code generated from the differential drive example is shown in Listing 10.3. This code is written to be compatible with numerical solvers provided by the *Apache Commons Math* library[3].

The generated class implements the interface *IFirstOrderModel*, which extends the interface *FirstOrderDifferentialEquation* defined in the *Apache Math* library. It defines the following three methods:

1. *computeDerivatives*, called by the solver, contains the definition of the state transition function (the state $\vec{x}$ is mapped on the array y, while the velocities $\dot{\vec{x}}$ are mapped on $yDot$);
2. *setParameter*, which can be used to set the parameters of a specific robot;

[3]Apache Commons Math - http://commons.apache.org/math/

```
1  package robotics.models;
2
3  public class DifferentialDrive implements IFirstOrderModel {
4
5          private double L, r, u_l, u_r;
6
7          public void setAction(double[] actions) {
8                  if (actions.length != 2)
9                          throw new IllegalArgumentException(``Actions must have size 2.'');
10                 u_l = actions[0];
11                 u_r = actions[1];
12         }
13         public void setParameters(double[] parameters) {
14                 if (parameters.length != 2)
15                         throw new IllegalArgumentException(``Parameters must have size 2.'');
16                 L = parameters[0];
17                 r = parameters[1];
18         }
19         public void computeDerivatives(double t, double[] y, double[] yDot) throws
               DerivativeException{
20                 yDot[0] = r / 2 * (u_l + u_r) * java.lang.Math.cos(y[2]);
21                 yDot[1] = r / 2 * (u_l + u_r) * java.lang.Math.sin(y[2]);
22                 yDot[2] = (r / L) * (u_r - u_l);
23         }
24 }
```

Listing 10.3: Differential Drive implementation

3. *setAction*, which can be used to set the actions.

The methods *setParameter* and *setAction* are called by the algorithm that uses the differential model (e.g. path planner or simulator).

10.2.3 Implementation details

The grammar of DCML was defined by means of AntLR3 [113]. It was used also for the definition of the semantic actions and for building the AST tree. AntLR is a parser generator that reduces the time and effort needed for building and maintaining language processing tools.

One of the problems of MDE is that developers, in order to use MDE techniques, usually need tools that support them in the management and development of models. The Xtext framework [114] was used for creating this tool. Xtext provides a simple way for creating textual DSLs and to automatically generate a full-featured Eclipse Text Editor from the grammar.

The grammar implemented in the Xtext editor is the same used for the parser, without semantic actions.

The parser and the editor were separately implemented for two reasons. First, thanks to AntLR it is possible to have a better control with respect to Xtext, on both the AST creation phase and on the definition of the syntax and the semantic of the language. Second, by using AntLR the developed tool can be used in a stand-alone way or can be integrated in others IDEs.

Xtext was used in order to integrate DCML in the Eclipse IDE. This integration gives to developers useful features such as auto-completion and syntax highlighting, while expressing the grammar using AntLR give us the power of expressing complex semantic rules.

In order to integrate the parser in the editor a new button that allows the invocation of the parser was defined in the Eclipse toolbar. The parser takes as input the model created by the user and validates it. Then, in the case that the model is correct, the Java translator is invoked. The translator takes the AST produced by the parser and translates it into the Java class that implements the differential equations of the model and conforms to the solver required interfaces.

10.3 Case study

Using DCML it is possible to describe models of simple robots, such as the differential drive described in Section 10.2, or models of more complex robots like BART.

BART is an omnidirectional holonomic wheeled robot, developed by the Software for Experimental Robotics Lab (SERL) at the University of Bergamo. It is made up of two steering blocks and two free wheels. Each block is a differential drive and the rotational joint is not on the axis of its wheels. The mechanical structure of the robot is presented in figure 10.3. BART is slightly similar to the robots presented in [115] and [116].

While modeling the kinematics of BART by defining the rigid bodies that made it up can be quite difficult, modeling the general kinematics of the robot can be done quite easily using differential models, and thus by means

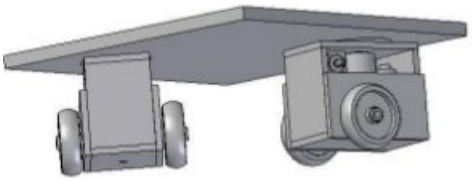

Figure 10.3: The BART robot

of DCML, as shown in Listing 10.4.

The BART configuration space is described in terms of the variables x, y and *theta* (cartesian position and the orientation of the center of the robot), the variables x_front, y_front and phi_front (cartesian position and orientation of the joint that connects the base of the robot to the frontal differential drive, related to the absolute reference system), and the variables x_rear, y_rear and phi_rear (position and orientation of the rear steering block).

The parameters of the model are $tt\_wheel\_half\_axis$, that represents the

```
1  BEGIN Bart
2   PACKAGE : robotics.models;
3   ACTION : left_f_s, right_f_s, left_r_s, right_r_s;
4   PARAM : tt_wheel_half_axis, tt_wheel_radius, tt_steer_offset;
5   CONFIG : x, y, theta, x_front, y_front, phi_front, x_rear, y_rear, phi_rear;
6   VAR : k;
7
8   k = tt_steer_offset / tt_wheel_half_axis;
9
10  d(x_front) = (tt_wheel_radius /2) * (((cos(phi_front) - k * sin(phi_front)) * right_f_s) + ((cos
        (phi_front) + k * sin(phi_front)) * left_f_s));
11  d(y_front) = (tt_wheel_radius /2) * (((sin(phi_front) + k * cos(phi_front)) * right_f_s) + ((sin
        (phi_front) - k * cos(phi_front)) * left_f_s));
12  d(phi_front) = (tt_wheel_radius / (2 * tt_wheel_half_axis)) * (right_f_s - left_f_s);
13
14  d(x_rear) = (tt_wheel_radius /2) * (((cos(phi_rear) - k * sin(phi_rear)) * right_r_s) + ((cos(
        phi_rear) + k * sin(phi_rear)) * left_r_s));
15  d(y_rear) = (tt_wheel_radius /2) * (((sin(phi_rear) + k * cos(phi_rear)) * right_r_s) + ((sin(
        phi_rear) - k * cos(phi_rear)) * left_r_s));
16  d(phi_rear) = (tt_wheel_radius / (2 * tt_wheel_half_axis)) * (right_r_s - left_r_s);
17
18  x = (x_front + x_rear)/2;
19  y = (y_front + y_rear)/2;
20  theta = atan2((y_front-y_rear),(x_front-x_rear)) + (pi / 4);
21 END;
```

Listing 10.4: BART model

half length between the wheels in one of the differential drives, tt_wheel_radius, that is the radius of each wheel of the steering blocks, and tt_steer_offset, that is the distance between the steering axis and the axis of the wheels of the robot. The variable k was introduced for representing the ratio between tt_steer_offset and $tt\_wheel\_half\_axis$ to simplify the writing of the differential equations. The actions accepted by the robot are the values $left\_f\_s$ and $right\_f\_s$, which represent the angular speeds of left and right wheels of the frontal steering block, and the values $left\_r\_s$ and $right\_r\_s$, which represent the angular speeds of left and right wheels of the rear differential drive.

The differential model of BART can be divided in three parts. The first one, that involves the x_front, y_front and phi_front variables, expresses the differential equations needed to compute the position of the joint of the front steering block. It is an extension of the differential model for a standard differential drive that considers also the fact that the rotational joint is not on the axis of the differential drive. The second part, involving the variables x_rear, y_rear and phi_rear, is quite similar to the first one but it is related to the rear steering block. The last part of the model, involving the variables x, y and $theta$, computes the position of the center of the BART robot. This part is not made up of differential equations because these values can be computed with algebraic equations from the values of the two steering blocks.

A Java framework that implements some well known algorithms for sampling-based path planning was implemented and integrated with DCML. All the provided algorithms depend on the model of the robot under simulation. For this reason they were designed in such a way that the algorithm implementation and the differential model are clearly separated. The differential model is thus an input parameter. Using DCML it is possible to modify the implementation of the differential model without manually changing any line of the source code. The only steps required are:

1. modify the DCML document,

2. regenerating from it the new source code.

This is possible thanks to the DCML to Java translator that was developed and that, taken as input the AST of the model, creates the class that implements the model itself. The integration between the framework and DCML was tested with two robot models: the differential drive and BART. A screenshot of a planning result is depicted in figure 10.4.

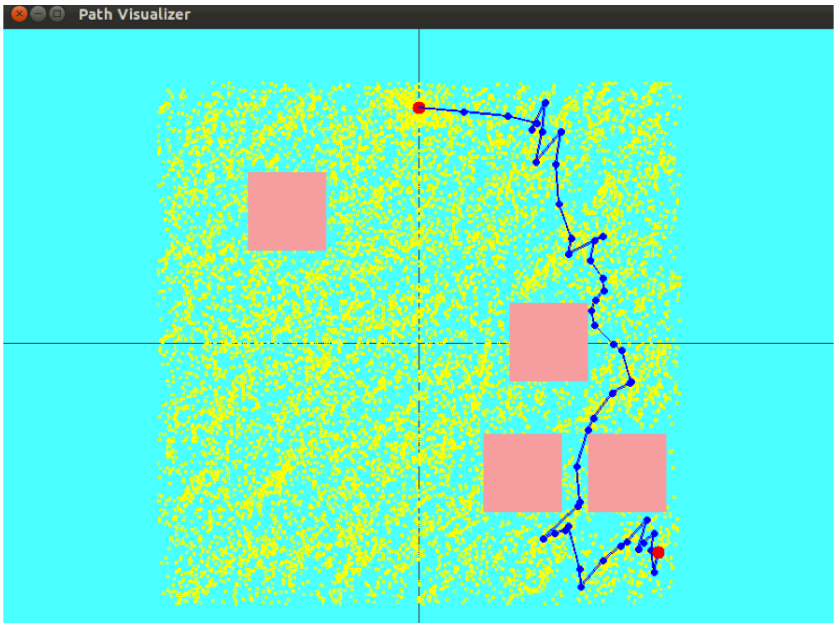

Figure 10.4: The integration of DCML with a Path Planning algorithm

10.4 Discussion

This chapter presented the Differential Constraints Modeling Language (DCML), a Domain Specific Language that allows the description of differential models with a high level of abstraction from implementation details. Using DCML users can focus only on modeling the state transition function of the robot. The source code can be then automatically generated by translators optimized according to both the destination programming language and the solver required interface. In this way the details related to the model implementation (e.g. the numerical solver used to solve the equations) can be completely hidden to the user and the task of creating optimized code can

be delegated to the writer of the translator. The presented approach allows developers to define new translators in order to generate code optimized accordingly to real-time and computational requirements.

This work shows that MDE can be used with good results in specific areas, such as the representation of differential models, in which the representation of the knowledge can be formalized in a defined model. The integration of the generated models in the path planning framework demonstrates that the technique can have useful application also in a robotics environment.

11 Conclusions

In this thesis I presented the research carried out during my PhD, which focused on investigating new approaches for the development of flexible component-based robotics software systems.

Chapter 2 presented a new contribution to the analysis of the variability that characterizes the robotics domain. In particular it describes how the variability of situatedness, embodiment, intelligence and software framework influences the design of robotics software systems.

In the next chapters I focused on the modeling and the resolution of the variability (part I) and on new approaches for the development flexible systems (part II).

The first part presented a Model-Driven Development Process for modeling and resolving the variability in component-based robotic systems. The proposed approach builds on the concept of Component Framework and is based on the definition of three sets of models and meta-models for representing different concerns. The Template System Model describes the architecture of a component-based software product line and explicitly represents the variation points, the Feature Model represents the variability in system functionalities, and finally the Resolution Model waves the first two models and describes how the variability should be resolved at deployment-time. Thanks to the orthogonality of the Template System Model and of the Feature Model, the system integrator is not required anymore to master software engineering concepts and technologies.

The proposed approach was applied for defining a set of flexible component-based software architectures for the robust navigation. These architectures compose the Robust Navigation Software Product Line. Unlike other robotics software libraries, the presented architectures are designed by taking into account the variability in robotic system requirements and the commonalities in existing open source libraries for mobile robots. The resulting Robust Navigation component-based product line can be easily configured according to the requirements of specific applications by using the models and the tools that I have developed.

Future works will try to improve the presented development process by addressing important robotics requirements like QoS awareness and runtime reconfiguration. A major reason for the limited ability of existing robotic systems to exhibit both high robustness and high versatility is the lack of mechanisms that allow the robot to trade between conflicting QoS requirements (i.e. performance vs. long-term continuous operation, completeness vs. dependability) when the actual operational context changes (tasks, environment, resources). These mechanisms would allow the robot to dynamically reconfigure its own control and coordination system in order to maximize overall QoS according to the actual operational context and available resources.

The exploitation of the approach presented in this thesis will allow the development of dynamically adaptive robotic systems, where robotic engineers can define several variation points (resources, algorithms, control strategies, coordination policies, cognitive mechanisms and heuristics, etc.) and depending on the context, the system dynamically chooses suitable variants to realize those variation points. These variants may provide better quality of service (QoS), offer new services that did not make sense in the previous context, or discard some services that are no longer useful.

Another topic that will be addressed in future works is the generation of configured system models that describe hybrid architectures, where components implemented by means of different software frameworks interact for building distributed and complex systems.

The first part of this document also described in details the concept of refactoring, which is one of the phases of the development process that aims

to refactor existing object oriented legacy code. The result of the refactoring is a flexible and software framework independent software library, which can be easily encapsulated in reusable components. In this direction the chapter 5 presented a set of guidelines and refactoring patterns and, by means of a case study, it demonstrated how they have been applied for refactoring the CoPP library.

In the second part of the thesis each chapter contains a "Discussion" section, which deeply elaborates the conclusions about the research and the problems addressed in that chapter. For this reason in the following paragraphs I only summarize the research topics.

In the seventh chapter I presented JOrocos, a software library that, by resolving a set of architectural mismatches, makes possible the cooperation between Service Oriented Architectures (SOA) and Data Flow Oriented Architectures. In particular it focuses on SCA and Orocos, the first a component based SOA and the second a hard real-time component based robotics framework.

In order to support JOrocos and the development of hybrid systems (i.e. systems where SCA/Java and Orocos/C++ component interact), in the eighth chapter I documented a performance comparison between Java and C++. This study shows that Java can be considered nowadays an alternative to C++, at least for the development of high-level and non real-time functionalities

In the ninth chapter I faced the problem of separating different concerns in the design of component based robotics systems. In particular I presented an approach based on SCA and ASM, which has the goal of demonstrating how the Computation and the Coordination concerns can be decoupled by separately modeling them. Thanks to this orthogonal separation, it is possible to implement different coordination policies, which orchestrate the execution of the services provided by the various components of the system, without having to modify the implementation of the services themselves.

Finally in the tenth chapter I demonstrated how a domain specific language can be designed and used for describing robot differential models and automatically generate the source code that implements them. The chapter shows how, thanks to this approach, it is possible to implement algorithms

(e.g. sample-base path planners) that are not hard-coupled to a specific robot model and are hence more flexible.

Bibliography

[1] DLR Justin. http://www.dlr.de/rm/en/desktopdefault.aspx/tabid-5471/8991_read-16694/, 2009. (Cited on page 1.)

[2] Fraunhofer IPA Care-O-bot. http://www.care-o-bot.de/english/Care-O-bot_3.php, 2008. (Cited on page 1.)

[3] Willow Garage PR2. http://www.willowgarage.com/pages/pr2/overview, 2008. (Cited on page 1.)

[4] N. Tomatis, "Bluebotics: Navigation for the clever robot," *Robotics & Automation Magazine, IEEE*, vol. 18, no. 2, pp. 14–16, 2011. (Cited on page 2.)

[5] S. Cousins, "Exponential growth of ros," *Robotics & Automation Magazine, IEEE*, vol. 18, no. 1, pp. 19–20, 2011. (Cited on page 2.)

[6] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. (Cited on pages 4, 24 e 32.)

[7] D. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006. (Cited on pages 4 e 24.)

[8] D. Brugali, W. Nowak, L. Gherardi, A. Zakharov, and E. Prassler, "Component-based refactoring of motion planning libraries," in *Intelli-*

gent Robots and Systems (IROS), 2010 IEEE/RSJ Int. Conference on, pp. 4042–4049, IEEE, 2010. (Cited on page 5.)

[9] D. Brugali, L. Gherardi, M. Klotzbuecher, and H. Bruyninckx, "Service component architecture in robotics: the sca-orocos integration," in *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, page to be submitted (ISoLA 2011)*, (Vienna, Austria), October 18-19 2011. (Cited on page 6.)

[10] L. Gherardi, D. Brugali, and D. Comotti, "A java vs. c++ performance evaluation: a 3d modeling benchmark," in *3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2012)*, (Tsukuba, Japan), November 5-8 2012. (Cited on page 6.)

[11] D. Brugali, L. Gherardi, E. Riccobene, and P. Scandurra, "A formal framework for coordinated simulation of heterogeneous service-oriented applications.," in *8th International Symposium on Formal Aspects of Component Software (FACS)*, 2011. (Cited on page 6.)

[12] M. Guarnieri, E. Magri, D. Brugali, and L. Gherardi, "A domain specific language for modeling differential constraints of mobile robots," in *12th International Conference on Autonomous Robot Systems and Competitions (Robotica 2012)*, (Guimaraes, Portugal), April 11 2012. (Cited on page 7.)

[13] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003. (Cited on page 9.)

[14] J. Poulin, *Measuring software reuse: principles, practices, and economic models*. Addison-Wesley, 1997. (Cited on page 10.)

[15] J. Radatz, A. Geraci, and F. Katki, *IEEE standard glossary of software engineering terminology*. IEEE Computer Society, 1990. (Cited on pages 10 e 11.)

[16] D. Kortenkamp, R. Simmons, and D. Brugali, "Robotic systems architectures and programming," *Springer Handbook of Robotics - Second Edition*, 2013, in press. (Cited on page 13.)

[17] M. Mataric *et al.*, "Situated robotics," *Encyclopedia of cognitive science*, vol. 4, pp. 25–30, 2002. (Cited on page 13.)

[18] I. Nesnas, "The claraty project: Coping with hardware and software heterogeneity," *Software Engineering for Experimental Robotics*, pp. 31–70, 2007. (Cited on pages 13 e 15.)

[19] R. Brooks *et al.*, "Intelligence without reason," *Artificial intelligence: critical concepts*, vol. 3, pp. 107–63, 1991. (Cited on pages 13 e 16.)

[20] D. Brugali and E. Prassler, "Software engineering for robotics," *Robotics & Automation Magazine, IEEE*, vol. 16, no. 1, pp. 9–15, 2009. (Cited on page 17.)

[21] D. e. a. Calisi, "Design choices for modular and flexible robotic software development: the openrdk viewpoint," *Journal of Software Engineering for Robotics*, vol. 3, no. 1, pp. 13,27, 2012. (Cited on page 17.)

[22] "ROS: Robot Operating System." `http://www.ros.org`, 2007. (Cited on page 18.)

[23] "Orocos: Open RObot COntrol Software." `http://www.orocos.org`, 2001. (Cited on pages 19 e 120.)

[24] "Service Component Architecture (SCA)." `http://www.osoa.org`. (Cited on pages 19, 120 e 154.)

[25] "EMF: The Eclipse Modeling Framework." `http://www.eclipse.org/modeling/emf/`. (Cited on pages 20, 39 e 64.)

[26] "Orocos RTT meta model." `http://www.best-of-robotics.org/bride/`. (Cited on page 20.)

[27] D. Brugali and P. Scandurra, "Component-based robotic engineering (part i)[tutorial]," *Robotics & Automation Magazine, IEEE*, vol. 16, no. 4, pp. 84–96, 2009. (Cited on pages 23 e 30.)

[28] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. (Cited on pages 27, 69 e 73.)

[29] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002. (Cited on page 28.)

[30] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *Robotics & Automation Magazine, IEEE*, vol. 4, no. 1, pp. 23–33, 1997. (Cited on pages 31, 97 e 113.)

[31] I. Ulrich and J. Borenstein, "Vfh+: Reliable obstacle avoidance for fast mobile robots," in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, vol. 2, pp. 1572–1577, IEEE, 1998. (Cited on pages 31, 97, 101 e 113.)

[32] K. Kang, "Feature-oriented domain analysis (FODA) feasibility study," tech. rep., DTIC Document, 1990. (Cited on page 32.)

[33] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, "Generative programming for embedded software: An industrial experience report," in *Generative Programming and Component Engineering*, pp. 156–172, Springer, 2002. (Cited on pages 35 e 36.)

[34] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, "Extending feature diagrams with UML multiplicities," in *6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA*, Citeseer, 2002. (Cited on pages 35 e 51.)

[35] "GMF, the Eclipse Graphical Modeling Framework." `http://www.eclipse.org/gmf/`. (Cited on page 39.)

[36] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008. (Cited on page 39.)

[37] P. Clements, F. Bachmann, L. Bass, D. Garlan, P. Merson, J. Ivers, R. Little, and R. Nord, *Documenting software architectures: views and beyond*. Addison-Wesley Professional, 2010. (Cited on page 44.)

[38] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration using feature models," *Software Product Lines*, pp. 162–164, 2004. (Cited on page 50.)

[39] "MVEL, MVflex Expression Language." `http://mvel.codehaus.org/`. (Cited on page 61.)

[40] "EMFT: Eclipse Modeling Framework Technology Project." `http://www.eclipse.org/modeling/emft/`. (Cited on page 64.)

[41] "EMF Feature Model." `http://www.eclipse.org/modeling/emft/featuremodel/`. (Cited on page 64.)

[42] "Pure::variants." `http://www.pure-systems.com/`. (Cited on page 64.)

[43] "Fmp: Feature Model Plug-in." `http://gsd.uwaterloo.ca/fmp`. (Cited on page 64.)

[44] "XFeature - Feature Modelling Tool." `http://www.pnp-software.com/XFeature/`. (Cited on page 64.)

[45] "FeatureIDE, an Eclipse plug-in for Feature-Oriented Software Development." `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`. (Cited on page 65.)

[46] "Common Variability Language (CVL)," August 13, 2012. OMG Revised Submission. Oystein Haugen et al. - IBM, Fraunhofer FOKUS, Thales and Tata Consultancy Services. (Cited on pages 66 e 67.)

[47] J. Lee, D. Muthig, and M. Naab, "A feature-oriented approach for developing reusable product line assets of service-based systems," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1123–1136, 2010. (Cited on pages 67 e 68.)

[48] E. Cirilo, I. Nunes, U. Kulesza, and C. Lucena, "Automating the product derivation process of multi-agent systems product lines," *Journal of Systems and Software*, vol. 85, no. 2, pp. 258–276, 2012. (Cited on page 68.)

[49] E. Cirilo, U. Kulesza, R. Coelho, C. de Lucena, and A. von Staa, "Integrating component and product lines technologies," *High Confidence Software Reuse in Large Systems*, pp. 130–141, 2008. (Cited on page 68.)

[50] M. Strandberg, "Robot path planning: An object-oriented approach," in *PhD thesis, KTH*, 2004. `http://sourceforge.net/projects/copp/`. (Cited on pages 69 e 79.)

[51] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005. (Cited on pages 70 e 73.)

[52] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns*. Morgan Kaufmann, 2002. (Cited on pages 73, 74 e 75.)

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. (Cited on pages 76, 91, 124 e 128.)

[54] M. Fowler, "Inversion of control containers and the dependency injection pattern," *Martin Fowler Web Site*, 2004. `http://martinfowler.com/articles/injection.html`. (Cited on page 77.)

[55] S. LaValle, *Planning algorithms*. Cambridge Univ Pr, 2006. (Cited on pages 78, 104 e 113.)

[56] I. Sucan and L. Kavraki, "On the implementation of single-query sampling-based motion planners," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 2005–2011, IEEE, 2010. (Cited on page 78.)

[57] "MSL - Motion strategy library - S. LaValle, P. Cheng, J. Kuffner, S. Lindemann, A. Manohar, B. Tovar, L. Yang, and A. Yershova." `http://msl.cs.uiuc.edu/msl/`. (Cited on page 79.)

[58] "MPK - Motion Planning Kit - J.C. Latombe, F. Schwarzer, and M. Saha." `http://robotics.stanford.edu/~mitul/mpk/`. (Cited on page 79.)

[59] I. Gipson, K. Gupta, and M. Greenspan, "Mpk: An open extensible motion planning kernel," *Journal of Robotic Systems*, vol. 18, no. 8, pp. 433–443, 2001. (Cited on page 79.)

[60] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, 2008. (Cited on page 79.)

[61] E. Plaku, K. Bekris, and L. Kavraki, "Oops for motion planning: An online, open-source, programming system," in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 3711–3716, IEEE, 2007. `http://www.kavrakilab.org/OOPSMP/index.html`. (Cited on page 80.)

[62] "OMPL - Open Motion Planning Library - Ioan Sucan." `http://www.ros.org/wiki/ompl`. (Cited on page 80.)

[63] T. Siméon, J. Laumond, and F. Lamiraux, "Move3d: A generic platform for path planning," in *Assembly and Task Planning, 2001, Proceedings of the IEEE International Symposium on*, pp. 25–30, IEEE, 2001. (Cited on page 80.)

[64] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer-Verlag New York Inc, 2008. (Cited on page 96.)

[65] J. Minguez and L. Montano, "Nearness diagram (nd) navigation: collision avoidance in troublesome scenarios," *Robotics and Automation, IEEE Transactions on*, vol. 20, no. 1, pp. 45–59, 2004. (Cited on page 97.)

[66] S. Quinlan and O. Khatib, "Elastic bands: Connecting path planning and control," in *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pp. 802–807, IEEE, 1993. (Cited on page 101.)

[67] S. Thrun, "Learning occupancy grids with forward sensor models," *Autonomous Robots*, vol. 15, pp. 111–127, 2002. (Cited on page 104.)

[68] T. Wiemann, A. Nuechter, K. Lingemann, S. Stiene, and J. Hertzberg, "Automatic construction of polygonal maps from point cloud data," in *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Workshop on*, IEEE, 2010. (Cited on page 104.)

[69] H. Kato and M. Billinghurst, "Marker tracking and hmd calibration for a video-based augmented reality conferencing system," in *Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on*, pp. 85–94, IEEE, 1999. (Cited on page 106.)

[70] D. Wagner and D. Schmalstieg, "Artoolkitplus for pose tracking on mobile devices," in *Proceedings of 12th Computer Vision Winter Workshop (CVWW'07)*, pp. 139–146, 2007. (Cited on page 106.)

[71] O. Mozos, Z.-C. Marton, and M. Beetz, "Furniture models learned from the www," *Robotics Automation Magazine, IEEE*, vol. 18, pp. 22 –32, june 2011. (Cited on page 119.)

[72] M. Tenorth, U. Klank, D. Pangercic, and M. Beetz, "Web-enabled robots," *Robotics Automation Magazine, IEEE*, vol. 18, pp. 58 –68, june 2011. (Cited on page 119.)

[73] T. Yoshimi, N. Matsuhira, K. Suzuki, D. Yamamoto, F. Ozaki, J. Hirokawa, and H. Ogawa, "Development of a concept model of a robotic

information home appliance, aprialpha," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 1, pp. 205 – 211, 2004. (Cited on page 119.)

[74] E. Marks and M. Bell, *Service-Oriented Architecture (SOA): A planning and implementation guide for business and technology.* John Wiley & Sons, 2006. (Cited on page 120.)

[75] R. de Molengraft, van van, M. Beetz, and T. Fukuda, "A special issue toward a www for robots," *Robotics Automation Magazine, IEEE*, vol. 18, p. 20, june 2011. (Cited on page 120.)

[76] "KUKA youBot store." `http://youbot-store.com/`. (Cited on page 121.)

[77] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003. (Cited on page 123.)

[78] "Rosjava - An implementation of ROS in pure Java with Android support." `http://code.google.com/p/rosjava/`. (Cited on pages 137 e 138.)

[79] "Open Scene Graph." `http://www.openscenegraph.org`. (Cited on pages 137 e 142.)

[80] M. Long, A. Gage, R. Murphy, and K. Valavanis, "Application of the distributed field robot architecture to a simulated demining task," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE Int. Conference on*, IEEE, 2005. (Cited on page 138.)

[81] S. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time Java for industrial robot control," in *Proceedings of the 5th Int. workshop on Java technologies for real-time and embedded systems*, pp. 104–110, ACM, 2007. (Cited on page 139.)

[82] F. Raimondi, L. Ciancimino, and M. Melluso, "Real-time remote control of a robot manipulator using java and client-server architecture," in *Proceedings of the 7th Int. Conference on Automatic Control, Modeling and Simulation*, 2005. (Cited on page 139.)

[83] A. Elnagar and L. Lulu, "A global path planning Java-based system for autonomous mobile robots," *Science of Computer Programming*, vol. 53, no. 1, pp. 107–122, 2004. (Cited on page 139.)

[84] F. Monteiro, P. Rocha, P. Menezes, A. Silva, and J. Dias, "Teleoperating a mobile robot. A solution based on JAVA language," in *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE Int. Symposium on*, vol. 1, IEEE, 2002. (Cited on page 139.)

[85] "Just in time compiler." `http://en.wikipedia.org/wiki/JIT_compiler`. (Cited on page 140.)

[86] S. Spw, S. Wentworth, and D. Langan, "Performance evaluation: Java vs c++," in *39th Annual ACM Southeast Regional Conference*, Citeseer, March 16-17 2001. (Cited on pages 140 e 141.)

[87] L. Bernardin, B. Char, and E. Kaltofen, "Symbolic computation in Java: an appraisement," in *Proceedings of the 1999 Int. symposium on Symbolic and algebraic computation*, pp. 237–244, ACM, 1999. (Cited on pages 140 e 141.)

[88] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey, "A methodology for benchmarking Java Grande applications," in *Proceedings of the ACM 1999 conference on Java Grande*, pp. 81–88, ACM, 1999. (Cited on pages 140 e 141.)

[89] M. Roulo, "Accelerate your Java apps," *Java World*, 1998. (Cited on pages 140, 141 e 149.)

[90] C. Mangione, "Performance tests show java as fast as c++," *JavaWorld*, 1998. (Cited on pages 140 e 141.)

[91] J. Lewis and U. Neumann, "Performance of Java versus C++," *Computer Graphics and Immersive Technology Lab, University of Southern California, Jan*, 2003. (Cited on page 141.)

[92] L. Prechelt *et al.*, "Comparing Java vs. C/C++ efficiency differences to interpersonal differences," *Communications of the ACM*, vol. 42, no. 10, pp. 109–112, 1999. (Cited on page 141.)

[93] B. Delaunay, "Sur la sphere vide," *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, pp. 793–800, 1934. (Cited on page 142.)

[94] "Open Scene Graph API reference." `http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs`. (Cited on page 142.)

[95] S. Wilson and J. Kesselman, *JavaTM Platform Performance - Chapter 8*. Sun Microsystems, 2001. `http://java.sun.com/docs/books/performance/1st_edition/html/JPAlgorithms.fm.html`. (Cited on pages 143 e 145.)

[96] "Specifications of java.util.collections." `http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html`. (Cited on page 144.)

[97] S. Meloan, "The Java HotSpot (tm) Perfomance Engine: An In-Depth Look," *Article on Sun's Java Developer Connection site*, 1999. (Cited on page 147.)

[98] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Proceedings of the Int, Workshop on Trends in Distributed Systems CORBA and Beyond*, pp. 162–176, Springer LNCS vol. 1161, 1996. (Cited on page 153.)

[99] E. Borger and R. Stark, "Abstract State Machines: A method for high-level system design and analysis," 2003. (Cited on pages 154, 163 e 164.)

[100] E. Riccobene, P. Scandurra, and F. Albani, "A modeling and executable language for designing and prototyping service-oriented applications," in *EUROMICRO-SEAA*, pp. 4–11, IEEE, 2011. (Cited on pages 154 e 166.)

[101] "The ASMETA toolset: an analysis toolset for ASMs," 2006. http://asmeta.sf.net/. (Cited on page 166.)

[102] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. Available at http://planning.cs.uiuc.edu/. (Cited on pages 181, 183, 184 e 185.)

[103] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, 2000. (Cited on page 182.)

[104] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, 2005. (Cited on page 182.)

[105] J. Laumond, S. Sekhavat, and F. Lamiraux, "Guidelines in nonholonomic motion planning for mobile robots," *Robot motion planning and control*, 1998. (Cited on page 183.)

[106] E. Szádeczky-Kardoss and B. Kiss, "Extension of the rapidly exploring random tree algorithm with key configurations for nonholonomic motion planning," in *Mechatronics, 2006 IEEE International Conference on*, IEEE, 2006. (Cited on page 183.)

[107] J. Riehl, "Implementing the myfem embedded domain-specific language," in *Proceedings of the Second International Workshop on Domain-Specific Program Development*, DSPD'08, 2008. (Cited on page 184.)

[108] J. A. Miller, J. Han, and M. Hybinette, "Using domain specific language for modeling and simulation: Scalation as a case study," in *Winter Simulation Conference*, 2010. (Cited on page 184.)

[109] M. Hohn, "A little language for modularizing numerical pde solvers," *Software: Practice and Experience*, vol. 34, no. 9, 2004. (Cited on page 184.)

[110] H. Liu and P. Hudak, "An ode to arrows," *Practical Aspects of Declarative Languages*, 2010. (Cited on page 184.)

[111] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. (Cited on page 187.)

[112] N. Wirth, "Extended backus-naur form (ebnf), 1996," *ISO/IEC*, vol. 14977, 1996. (Cited on page 187.)

[113] "ANTLR - ANother Tool for Language Recognition." `http://www.antlr.org/`. (Cited on page 192.)

[114] "Xtext - Language Development Made Easy!." `http://www.eclipse.org/Xtext/`. (Cited on page 192.)

[115] F. Han, T. Yamada, K. Watanabe, K. Kiguchi, and K. Izumi, "Construction of an omnidirectional mobile robot platform based on active dual-wheel caster mechanisms and development of a control simulator," *J. Intell. Robotics Syst.*, vol. 29, 2000. (Cited on page 193.)

[116] T. Yamada, K. Watanabe, K. Kiguchi, and K. Izumi, "Dynamic model and control for a holonomic omnidirectional mobile robot," *Auton. Robots*, vol. 11, 2001. (Cited on page 193.)