



Classification and Integration of Software Component Models for Robotics

Andrea Luzzana

Thesis

Submitted to the Università degli Studi di Bergamo
for the degree of
Doctor of Philosophy

Advisor: Prof. Davide Brugali

Ph.D. course in Mechatronics, Information Technology,
New Technologies and Mathematical Methods
February 2013

*To Renata, Maurizio,
Paolo and Chiara*

Abstract

The Component Based Software Engineering (CBSE) allows the realization of modular and flexible systems composed by reusable software components. In the robotics field, the CBSE principles struggle to become spread used due to the extreme variability in functionality, applications and involved hardware that characterize the domain. This variability directly impacts on the component implementations limiting their reuse capability. The component model concept plays a fundamental role since it defines the rules that supervise the components definitions and their interactions. These rules strongly influences the capability of components to cope with the variability of the robotics domain.

The high number of proposed component models, both in literature and in the industrial field, claims for a model for their comparison and classification in order to highlight commonalities and differences among different approaches. In this work we propose a feature-based classification model that exploits a clear separation of concerns when analyzing different component models. The four identified concerns refer to the Communication, the Computation, the Configuration and the Coordination aspects. This approach allows for a better understanding of the constructs provided by different models in order to manage the variability of the system.

From the classification, it results clear that robotic domain specific models offer constructs that are well suited for addressing the specific issues of

robotics although they lack in providing a sufficient level of flexibility. On the other hand, general purpose models offer a higher level of flexibility without offering relevant robotic specific features. We propose the integration of component models as a key to overcome this limitation. In particular, the integration between the Service Component Architecture (SCA) and the Robot Operating System (ROS) allows the developers to fully exploit the benefits of both approaches while mitigating their deficiencies. We present the control software of the BART robot as a proof of the benefits of this approach.

Acknowledgements

There are so many people that I would like to thank for helping me in the past three years that I will surely forget someone.

The work presented in this thesis would not have been possible without the guidance and the encouragement of my supervisor prof. Davide Brugali, my thanks goes to him for giving me the opportunity to be a Ph.D. student.

My deepest thanks are for Renata and Maurizio, my parents. Thank you for having supported me during these years and for constantly guiding me through life. A special thank goes to my brother Paolo who has always been close to me.

I would like to thank my girlfriend Chiara. I am extremely lucky since I can rely on a person who can understand me, often before I speak, and with whom I can share joys, and sometimes sorrows, of life.

I cannot forget my companion in this adventure Luca, I wish you the best for your future and I am sure that you will reach great results. I am grateful also to Aldo for his help and for his friendship.

Finally, I would like to thank all the people who have offered me their help over these years. Among them, the staff of the laboratories of the university and, in particular, Ercole, Sergio, Daniele and Luca.

Andrea Luzzana
Dalmine, February, 2013

Contents

1	Introduction	1
1.1	Component Model for Robotics	3
1.1.1	Challenges in Robotics Software Engineering	4
1.2	The Proposed Approach	6
1.3	Structure of the Thesis	7
1.4	Acknowledgements	8
2	The “4C” Classification Model	9
2.1	Related Work	10
2.2	Feature Models	15
2.2.1	Basic Feature Models	16
2.3	Feature-based 4C Classification	21
2.3.1	Separation of concerns	21
2.3.2	Communication	23
2.3.3	Computation	30
2.3.4	Configuration	37
2.3.5	Coordination	45
3	Component Models Survey	51
3.1	Corba Component Model (CCM)	52
3.2	Fractal	54

3.3	Koala	56
3.4	KobrA	58
3.5	OpenCOM	59
3.6	OROCOS	61
3.7	Orca	64
3.8	Pin	67
3.9	Pecos	69
3.10	Robot Operating System	70
3.11	Rubus	72
3.12	SaveComp Component Model	74
3.13	Service Component Architecture	76
3.14	Smartsoft	79
3.15	SOFA 2.0	81
4	Component Models Analysis	85
4.1	Communication	86
4.1.1	Interface	86
4.1.2	Anonymity	90
4.1.3	Synchrony	92
4.2	Computation	93
4.2.1	Component Implementation Language	94
4.2.2	Component Behaviour	95
4.2.3	Real-time Support	96
4.2.4	Distributed System Support	96
4.3	Configuration	98
4.3.1	Configurable Entities	98
4.3.2	Configuration Model	100
4.3.3	Composition Mechanism	103
4.3.4	Persistence Mechanism	104
4.4	Coordination	105
4.4.1	Connector Feature	105
4.4.2	Component Roles	106
4.5	General Considerations	107

5	SCA and ROS Integration	113
5.1	The Service Component Architecture	115
5.1.1	Basic Principles	117
5.1.2	The SCA component model	118
5.1.3	Domains	119
5.1.4	Components	121
5.1.5	Services	122
5.1.6	References	123
5.1.7	Properties	124
5.1.8	Bindings	124
5.1.9	Composite	125
5.1.10	Wires	125
5.1.11	SCA Run-time Environments	126
5.2	The Robot Operating System	133
5.2.1	The ROS Component Model	135
5.2.2	ROS tools	141
5.3	Integration of SCA and ROS	144
5.3.1	Rosjava	146
5.3.2	The RosGate Component in ROS	147
5.3.3	The RosGate Component in SCA	150
6	The Bart Robot	155
6.1	Mechanical Structure	156
6.1.1	Motors and Encoders	157
6.1.2	Kinematics	158
6.2	Electronics and Computation devices	161
6.2.1	Microcontrollers	162
6.2.2	Embedded PC	164
6.3	Control Software	166
6.3.1	Slave Microcontroller Firmware	168
6.3.2	Master Microcontroller Firmware	169
6.3.3	Embedded PC Software	174
6.3.4	Remote Workstation Software	177

7 Conclusions	189
7.1 Future Work	191
Bibliography	193

List of Figures

2.1	Feature diagram of a car.	16
2.2	Feature modeling symbols.	20
2.3	Top-level feature diagram for component models classification.	22
2.4	The Communication feature diagram.	23
2.5	The Computation feature diagram.	31
2.6	The OROCOS finite state machine.	33
2.7	The Configuration feature diagram.	38
2.8	The Coordination feature diagram.	47
3.1	The Corba Component Model (CCM).	53
3.2	A generic Fractal component.	55
3.3	An OpenCOM capsule example.	61
3.4	The structure of an OROCOS component.	62
3.5	The internal FSM of an OROCOS component.	63
3.6	An example of the Orca ServerPush communication pattern.	66
3.7	Pin component overview.	68
3.8	SaveCCM features summary.	74
3.9	A SaveCCM system example.	77
3.10	SCA components and composites examples.	78
4.1	Communication summary table.	109

4.2	Computation summary table.	110
4.3	Configuration summary table.	111
4.4	Coordination summary table.	112
5.1	A SCA composite example.	119
5.2	A SCA domain example.	120
5.3	SCA graphical notation for components	121
5.4	Structure of a SCA run-time.	127
5.5	The Apache Tuscany run-time architecture.	128
5.6	The structure of a FraSCAti container.	130
5.7	A typical ROS network configuration.	134
5.8	Communication between two ROS nodes.	139
5.9	A ROS graph node.	141
5.10	An example of a ROS-Gazebo integrated system.	144
5.11	Conceptual view of the SCA-ROS integration.	145
5.12	UML class diagram of the <code>RosGate</code> component.	148
5.13	Integration between SCA and ROS systems.	151
6.1	The BART robot.	157
6.2	The kinematics scheme of the BART robot.	159
6.3	The overall electronic architecture of the BART robot.	165
6.4	The controllers architecture.	171
6.5	The structure of the payload of Canbus messages.	173
6.6	Serial communication messages.	173
6.7	Graph node of the supervisor component.	177
6.8	The composite diagram of the ROS-SCA integrated system.	178
6.9	The ROS graph node of the ROS-SCA integrated system.	182
6.10	The <code>RosGateSCA</code> component class diagram.	183
6.11	The <code>BartClient</code> component class diagram.	184
6.12	The GUI provided by the <code>BartClient</code> component.	187

The Component Based Software Engineering (CBSE) [76] [53] is an approach for designing and developing software artifacts that has arisen in the software engineering field in the last years. Its goal is to promote the realization of software systems by composing reusable off-the-shelf and custom-built software components. This can be considered to be partially in contrast with the traditional developing process which is based on well established phases for requirements analysis, system designing and implementation.

The CBSE approach can dramatically reduce the amount of code that needs to be written, and tested, by the designers of the software applications. The result is an improved software development process both regarding the development and test time and the overall quality of the software.

The reuse of the same component in a large number of different and heterogeneous systems by several developers makes the knowledge about the component usage, its efficiency, its reusability and its robustness available to a potentially large community of users and developers. This exchange of knowledge can greatly improve the overall quality of developed software components.

The definition provided by the IEEE Standard Glossary of Software Engineering Terminology [67] for *reuse* is “*pertaining to a software module or other work product that can be used in more than one computer program or software system*”. Hence, in software engineering, the reuse of software is a concept that is older than, and independent from, the CBSE approach.

Anyhow, the component-based approach has contributed to radically change the reuse approach form from *opportunistic* to *systematic*.

Following the opportunistic approach, the software engineer reuses previously developed pieces of software implementations that fit the current problem, typically by reusing libraries or single portions of code. On the other hand, when the systematic approach is exploited, the research and development team puts explicit effort in developing software modules that meet the requirements of a family of related applications. The developed modules, also called components, can then be used and reused to solve a larger class of problems.

The Component Based Software Engineering approach is founded on the concept of *component* that is, according to Szyperski's definition: "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*" [76].

This definition, although being broadly accepted, does not take into consideration another fundamental concept: the *component model*. A component model defines specific interaction and composition rules that guarantee the interoperability among components that are developed and used by different subjects. Keeping in mind the role of the component model, components can be classified as "*software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*" [53].

Component models play a fundamental role in defining component structure and interactions. The greatest part of the functional and non-functional characteristics of a component-based software depend from, or are related to, the adopted component model. In particular, the component model has a direct impact on the flexibility feature of the system, that is, the ease with which a component can be modified for use in applications or environment different from those for which it was designed.

1.1 Component Model for Robotics

In the robotics field there is a strong need to improve the software reuse for the construction of robotic software applications. Software reuse allows researcher and developers to share knowledge about specific fields of the robotics domain without the need to rewrite each other's code continuously for adapting it to different applications. For example, SLAM (Simultaneous Localization And Mapping) researchers can rely on navigation and obstacle avoidance functionality encapsulated into third-party components without the need of rewriting yet another obstacle avoidance algorithm.

The presence of a shared and well tested collection of components provides to the researchers the possibility to test their newly implemented algorithms on a common benchmark basis. This results in achieving more objective, accurate and repeatable results.

Nevertheless, the software reuse in the robotic field is not a wide spread approach. Most robotic research projects are based on custom software architectures and the greatest part of software is developed from scratch each time. As a result, valuable robotic applications are developed as a unique monolithic software with very poor possibility for reuse of implementations or algorithms.

These software applications, while offering good algorithms and solutions to common problems, are often not reusable even in slightly different applications or scenarios. This is due to the tied coupling between the particular software implementation and the robot hardware, the computational units and, above all, because of the use of a non-shared architecture for creating and composing software components.

The presence of a common underlying specification for component creation and interaction can greatly improve the reusability of implementations as well as the communication and the information sharing among different research communities.

Since specifications and rules for component creation and interactions are mainly related to the component model, the choice of the most suitable model is critical. However, this choice is made more difficult for two reasons. The

first is the presence of many alternative and, potentially suitable, component models. The second is the particular nature of the robotic domain.

1.1.1 Challenges in Robotics Software Engineering

A number of issues make the design, the realization and the deployment of software component models for robotics a challenging task. Robotics is an experimental science that involves several different disciplines ranging from mechanics, electronics, control theory, computer science and artificial intelligence. For this reason, software component models embody functionality belonging to different fields which must cooperate for achieving a common goal. Moreover, the technologies on which those different fields are based are quickly changing due to the relentless progresses in information technology and electronics.

The research in robotics pursues the goal of studying artificial systems, focusing in particular in their behaviour and in their interactions with various environments that span from industrial production cells to outdoor or domestic environments. This means that robots are experimented and used in many heterogeneous use cases and scenarios with different functional and non-functional requirements.

In order to be reusable, the software for robotics needs to be *flexible* to effectively cope with the extreme variability in environments, in functionality, in applications and in hardware-related technologies. The key to achieve flexibility is the possibility to predict the source of variability and analyze their impact on the software components.

One of the major source of variability is represented by the *hardware variability*. Robotic systems can have onboard computing devices that may include microcontrollers, programmable logic devices, PLCs, embedded computers and so forth. These devices have to interact with heterogeneous sensors and actuators in order to carry out their goals. Moreover, embedded computational hardware is often constrained for what regards computational power, current consumption, physical dimensions, weight and storage capability. For this reason, several offboard computational nodes are typically added in order

to execute computation intensive portions of the robotic task. This opens for new issues related to the networking technology variability and the data transfer among nodes.

Robots are artificial agents that are ultimately meant to be integrated into human environments. This expose the robotic systems to the extreme variability that is typical of humans environments. This *application variability* deeply impacts on the features required to the robotic control software. By changing the target environment of a robot, functional and non-functional requirements can change as well. As an example, we can consider the deep differences between an industrial robot working into a controlled and well structured environment, e.g. a production cell, and a domestic robot, e.g. a commercially available robotic vacuum cleaner, that need to operate in a non-structured, non-controlled and time changing domestic environment that, surely, was not originally intended to accommodate robots.

Typically, the overall robot functionality is achieved by the cooperation of many other functionality such as perception for obstacle avoidance and map building, localization, navigation, motion planning, motion control and so forth. Each functionality is subjected to different timing requirements, different functional, and above all non-functional, requirements. Many functionality are intrinsically related to real-time requirements.

Different pieces of the overall functionality rely on sensors that are also exploited by other functionality leading to non-trivial synchronization issues regarding the concurrent and safe access to shared resources. This scenario can be even more challenging if real-time constraints are taken into account. On a mobile manipulator robot, for example, mapping algorithms can rely on laser scanners and digital camera sensors for map building. The same cameras can be exploited by a visual servoing algorithm for controlling the manipulator and by the localization algorithms for estimating the position of the robot in the environment. This *functional variability* can have a deep impact on the software components features and on their design.

1.2 The Proposed Approach

To maximize the reusability of component implementations, software components should be designed having in mind the variability sources [36] as illustrated above. This advocates the separation of design concerns into a set of orthogonal dimensions. We refer to this as the “*separation of concerns*” principle. This clear separation allows to effectively analyze how variability affects the different concerns involved in component design.

The proposed approach aims to improve the reuse-oriented development of component-based software by clarifying the role of the component model in the definition of component construction and interaction rules. For this purpose we propose the “4 Concerns (4C) classification” with the target of highlighting the four orthogonal concerns that characterize the component design and finding how different component models face these concerns. The identified concerns are the *Communication*, the *Computation*, the *Configuration* and the *Coordination*.

The proposed classification scheme separately threat each one of the four concerns, or dimensions, for each analyzed component model by analyzing the mechanisms and the constructs provided by the component model according to each dimension.

The classification is based on the *Feature Models* approach that provides a mechanisms for clearly represent concepts, features and the relationships among them. The proposed classification is aimed to be both a mechanism for comparing different component models and an instrument for choosing the most appropriate model according to a particular set of requirements.

As already explained, a great part of the capability to reuse software components directly comes from the characteristics of the adopted component model, for this reason, we think that the choice of a component model, in the early stages of the system development, represents a critical phase. This choice can be difficult because of the great variety of available component models, for this, we argue that the presence of a compact, and yet complete, analysis and classification strategy could greatly help in the comparison, and in the choice, of the most appropriate component model.

Regarding the particular field of the robotics, we will conclude that domain-specific component models often lack in the needed flexibility for achieving a good reusability while general purpose component models do not offer some of the key requirements that are mandatory for robotics applications.

We argue that, up to now, and probably in the future, a component model that can meet all possible requirements cannot exist. In our opinion, the key for achieving the best performances in terms of flexibility, and consequently reusability of implementations, is the *integration* of different component models with the aim of exploiting each other strengths while mitigating their defects.

To this purpose we propose the integration between the Service Component Architecture (SCA) and the Robot Operating System (ROS) environments. We demonstrate the feasibility of this approach and the achieved improved flexibility by showing the results obtained during the development of the control software of a new experimental robot, the BART. In particular, the flexibility of a general-purpose component model, such as SCA, can be exploited for the design and the realization of higher-level functionality of the system. On the other hand, we can take advantage of the domain specific features of ROS when we need to design and implement lower-level and hardware related functionality.

1.3 Structure of the Thesis

The thesis is structured as follows: in Chapter 2 we show the adopted classification model, in particular, in Section 2.1 we analyze similar classifications presented in literature while in Section 2.2 we describe the feature models formalism. In Section 2.3 we present the proposed classification scheme.

In Chapter 3 we review a number of component models available in the community. Five of these models, considered as the most representative, are analyzed and classified in Chapter 4 by means of the proposed classification strategy.

In Chapter 5 we propose and explain the integration between SCA and ROS component models and in Chapter 6 we present the experimental results

of this integration by analyzing the software structure of the BART robot.

Finally, in Chapter 7 we summarize the results of the work and we outline some possible future directions.

1.4 Acknowledgements

The research leading to this thesis 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.

The “4C” Classification Model

In this chapter a classification model to capture a wide spectrum of component models is presented. To this purpose, *feature models* [71] are used to compare different component models and provide a survey and categorization of a number of existing approaches as presented in Chapters 3 and 4. Feature models document the result of applying the domain analysis technique to existing component models.

We propose to separate the analysis in four dimensions or *concerns*, the so called “4Cs”, namely: *Communication*, *Computation*, *Configuration* and *Coordination* to better focus on the features that make a component model different from another and, consequently, some component models more suitable to some contexts and not to other.

The four concerns consist in the “4Cs” already presented in [68] regarding generic component-based software systems and in [36] focusing on the field of the component-based software engineering for robotics control software.

The use of the feature models formalisms represent the main novelty in the classification approach that is being presented.

In Section 2.1 we briefly inspect other proposed classification approaches presented in literature, in Section 2.2 we give an introduction on the basic principles of the feature models and feature diagrams, then, in Section 2.3, the classification approach and the feature diagrams developed for the classification are presented and explained.

2.1 Related Work

The component models embody the essence of the component-based software engineering since the component model defines the rules to build components and compose them into software systems. Many analysis techniques, taxonomies and classification strategies have been applied to component models in literature in the last decade. The need for a classification usually comes from the fact that component-based software design and development failed in providing standard principles for the design, the development and the reusability of software. Other approaches, such as the object-oriented development, has, on the contrary, succeeded in this task being now well established in several fields.

In the greatest part of the classification approaches, the main underlying objective is to achieve a classification of the many available component models in order to successfully compare them. It is well known that, beside genuine general purpose component models, a great number of application-specific models has been developed addressing the issues of a specific field. Application specific component models can be found in the field of the embedded systems, telecommunications, Internet-based applications, database applications and robotics, to name a few.

Given the number of developed models and the great variety of solutions among them, a good classification model should be general enough to give instruments to compare and classify them but also specific enough to capture and highlight the differences and the particularity of each approach. The extreme variability in the component models is transposed in a great variability in the classification approaches proposed in literature.

The classification presented in [59] is focused on an idealized component life-cycle based on a widely accepted set of desiderata. These are given as follows: first, components should be preexisting reusable software units that developers can reuse to compose software. Second, components should be produced and used by different parties. Third, it should be possible to copy and instantiate components so that their reuse can be maximized. Fourth, components should be composed into composite components that, in turn,

can be composed into even larger composites or subsystems and so on.

Since the component model defines the rules to construct and compose components, it is considered crucial to identify the key prerequisites to meet the the above mentioned desiderata. Authors believe that the best starting point is the study and the classification of current component models. In the presented paper, thirteen major component models have been analyzed and classified into a taxonomy with four categories according to the composition phase of the idealized component life-cycle. The composition phase is intended to be the most important phase in the whole development process. In particular, the analysis puts emphasis on the presence and the management of a repository of components.

The first category is *design without repository* in which, in the design phase, no repository is available and components are all constructed from scratch. Examples of this category of component models are PECOS and Fractal. The second category is *design with deposit-only repository* in which a repository is available in the design phase but it does not offer support for component retrieval. This category includes EJB, COM, .NET and CCM component models. The third category comprises component models that offer a repository from which already-developed components can be retrieved in the deployment phase (*deployment with repository*). The last category is the *design with repository* in which a repository is available to store and retrieve components in the design phase. Koala, SOFA and KobrA belong to this category. Authors put into correlation these four categories with the different nature of the component concept implied in the analyzed component models (component as *objects* or as *architectural configurations*) and with the different language adopted for the definition of components (object-oriented programming language or ADL-like language).

The authors state that the ideal model should allow a repository-based composition mechanism both in the design phase and in the deployment phase and it does not exist yet. Encapsulation of functionality and an improved ability to compose components are identified as key features for an ideal component model.

This work, when compared with the approach that is being presented, fo-

cuses almost exclusively on the design and deployment phases of a component-based software system. Component models are analyzed and classified according to the instruments that they offer to support the developers in designing and putting together components, the availability and the usability of a repository seems to be the most important identified key-feature to ease the software development and reuse. On the other hand, the proposed analysis disregards what a specific component model offers in terms of constructs and artifacts to the developer and does not highlight the relation between these artifacts and the reference application domain of the component model (e.g. embedded systems or web-based applications).

In [56] the focus is on relating research results to relevant CBSE aspects and software domains. The authors' aim is to promote and enhance a systematic information exchange between researchers and adopters of CBSE technology through an on-line forum promoted by the European Union with the CBSEnet project. The goal of this project is to suggest how CBSE technologies could improve software engineering in different domains and to propose future research requirements for the development and deployment of CBSE technologies. To this purpose, a classification model has been developed in order to relate CBSE aspects, software domains and relevant research results to a set of reviewed component models. Six types of research results are identified: conferences and journal articles, technical reports, white papers, books and tools. The model takes into account eight CBSE aspects comprising *concepts, processes, roles, product concerns, business concerns, technology, off-the-shelf components* and *related development paradigms*. These aspects have been decomposed into more than fifty sub-aspects and they have been put into relation with sixteen application domains ranging from avionics to embedded systems and utility software. Among the CBSE aspects taken into account and chosen for the proposed component model classification there are: the component and the component model concept, contracts among components, interfaces, services and architectures. A particular emphasis is placed on the component-based development process which is split into the concepts of development *for* reuse and development *with* reuse. The results of the adoption of the classification model are presented thorough tables

showing, for each reviewed component model, its impact in terms of research results, the addressed CBSE aspects and the involved software domains.

Compared to the approach described in this work, the classification proposed by Kotonya et al. is strongly focused on the impact of component models on the research community in terms of research results. The proposed rich classification framework is complete and succeeds in addressing all the most relevant features of a component models from the available constructs to the development cycle. However, a clear separation of concerns between the analyzed concepts seems lacking.

The classification proposed in [46] aims to provide a complete framework to classify both general purpose and embedded systems specific component models. The classification is carried out by first identifying the basic principles of component models and then designing a framework to differentiate and classify component models. The classification is divided into four dimensions, each one of them takes into account both component specification, expressing component functions and extra-functional properties, and inter-component communication. This last feature is usually non-explicitly specified although different types of communication mechanisms are assumed in component models.

The first dimension is the *lifecycle* of components and it identifies the support provided by the component model during the lifecycle of the components or the component-based system. The *construct* dimension identifies both the component interfaces and the mechanisms of component binding and communication. The third dimension is represented by the *extra-functional properties*, here the authors collect the specification and support for the provision of property values to components and means for composition and management of these properties. The *domains* dimension is related with the application and business domains in which component models are used or supposed to be used.

Each one of the above mentioned dimensions is then analyzed and divided into a set of sub-dimensions allowing for a complete, and in some cases cumbersome, classification framework. The framework, however, identifies the minimal criteria for considering a model as a component model and allows

to recognize some recurrent patterns and similarities between component models that are developed and used for similar, or strictly related, application domains.

This approach is similar to our proposed classification to some extent, the proposed classification framework considers the separation of concerns as a key feature to compare and classify component models. The resulting framework is complete and the four axes of analysis are orthogonal though they differ from the classification dimension chosen in our work.

In [42] the focus is shifted on the component models for robotic control systems in order to extract a set of desirable features and compare existing models with a proposed component model (openRDK) specific for robotics applications. The analysis is focused on the component models that ensure modularity and rapid development tools for distributed robotic systems.

The classification model applies the separation of concerns, in particular, the analysis axes are: *concurrency models*, *information sharing techniques* and available *programming and debugging tools*. Concurrency models are related to the approach adopted for the concurrent execution of software modules or components across the possibly distributed computational nodes involved in the execution of the control of a robot. According to this classification, components can be treated as communicating independent processes, as threads inside a single process or as scheduled call-backs functions repeatedly called in response to some events or periodically. Information sharing mechanisms are strictly related to the concurrency model and deals with the information sharing among software modules. On the basis of this classification, components can centralize the exchange of data in a shared entity (*blackboard approach*) or can read and produce data through a set of input and output ports. According to the authors, the implementation of these paradigms usually rely on shared memory mechanisms or inter-process communication services. The last analysis axis take into account the presence, or the lack, of tools that can speed up the development of robotic software components such as on-line and off-line configuration utilities, specific debuggers and data loggers, robotic simulators and tools that enable the interoperability with other environments or operating systems.

Although the analysis is restricted to the very specific field of the component based robotic control software, the proposed approach is interesting for the separation of concerns adopted to separate the analysis into a set of orthogonal dimensions allowing for a better classification, and consequently a better understanding, of component models.

2.2 Feature Models

A *feature model* [71] is a hierarchical composition of features. In general, a *feature* is a distinguishable characteristic of a concept that is relevant for the concept itself. A concept can be an entire system, for example a software system, or a component. In the software engineering domain, feature models were proposed for the first time in 1990 in the context of the Feature Oriented Domain Analysis approach (FODA) [55]. The FODA approach aims to identify functionality and properties of a domain-specific software making a clear distinction between features that are present in all applications and features that are present only in some applications. For the representation of these functionality and properties, FODA proposed the *Feature Models* formal method. In this context, a feature is a software property representing an increase in software functionality.

In general, the feature models approach can be useful for classification or taxonomy purpose. Given a particular domain, that represents the *concept* of the feature model, a feature model can be built to represent all the possible *configurations* of the domain by representing the set of features that can, or cannot, be present in each configuration and the relationships among them. Following this approach, we used feature diagrams to classify the component models in Chapter 4.

A feature model that defines features, relations and dependencies can be graphically represented with a diagram that is typically a tree. We refer to this graphical notation as *feature diagram*.

Since its introduction, the original feature model formalism has been extended and several variants of the same formalism are now available in literature [71]. All these extensions do not impact on the core concerns of the

feature models approach but they attempt to compensate purported ambiguity or lack of precision and expressiveness of the original FODA featured models. Despite this, the original FODA feature diagrams have a formal semantics that has been demonstrated to be precise, unambiguous and expressively complete [31]. It is to remark that, in some cases, the proposed extensions simply aim to enrich the semantics of the formalism in order to be more effective in the representation of some specific domains. Notable examples of these extensions are the *feature cardinality* approach[47] and the *containment cardinality* approach.

Also the graphical notation of the feature diagrams has been adapted to coherently represent the extensions to the original formalism or to better highlight some particular aspects. We are now introducing a specific graphical formalism, also known as *basic feature models*.

2.2.1 Basic Feature Models

In this section we will refer to the feature diagram depicted in figure 2.1 in order to exemplify the characteristics of feature models. Features are

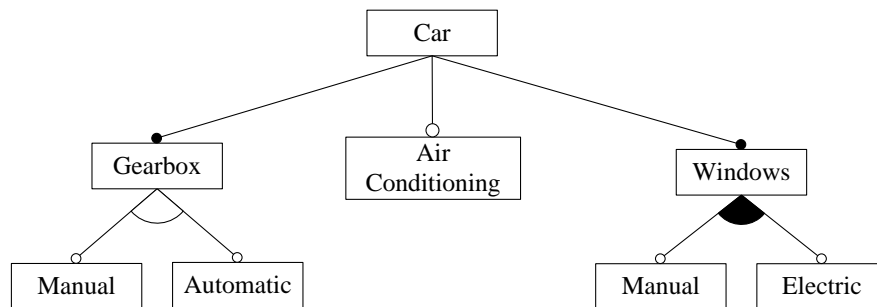


Figure 2.1: Feature diagram of a car.

represented as white boxes containing the feature name. Each feature is connected to its children features by an edge that represents the containment relationship. Feature diagrams are organized as a tree, where the root feature, also called *concept*, defines the domain that is modeled by the feature model. Features can be divided into two main categories according to their presence

in all possible configuration of the modeled domain:

- **Mandatory.** Mandatory features *have to* be present in every possible configuration of the modeled domain. They represent the core functionality or properties that characterize the modeled domain. In the example depicted in Figure 2.1, the features “Gearbox” and “Windows” are considered mandatory in every possible realization of a “Car”. It means that it cannot exist a car without the gearbox or without the windows. In the adopted graphical notation for feature diagrams, the mandatory features are represented by a black circle on the top of the feature box.
- **Optional.** Optional features *can* be present in some configurations of the modeled domain but they are not mandatory. They represent properties, functionality or characteristics that are specific to the particular configuration of the domain. In the example depicted in Figure 2.1, the feature “Air Conditioning” is optional. It means that particular configurations of the domain, in this case a particular model of a car, can have the air conditioning system while other configurations have not. In the feature diagrams graphical notation the optional features are represented by a white circle on top of the feature box.

Features can also be classified according to the nature of the containment relationship between a particular feature and its children. We can identify three types of containment relationships:

- **One-to-one.** The One-to-one containment means that the parent feature can contain the child feature. In particular, if the child feature is mandatory, this containment relation indicates that the parent feature *has to* contain the child feature, otherwise, if the child feature is optional, it simply indicates that the parent feature *can* contain the child feature. In the example in Figure 2.1, a “Car” *has to* contain a “Gearbox” and the “Windows” and it *can* contain the “Air Conditioning” feature. In the feature diagrams notation, this kind of containment relationship is depicted as a simple edge from the parent feature to its children.

- **OR.** The OR-containment represent a containment relationship between the parent feature and a set of children features. It means that the parent feature has to contain *at least one* of the children features. In the example, the relationship between “Windows” and its children “Manual” and “Electric” is an **OR** containment in the sense that car windows can be both manual and electric. According to the semantics of this relationship, a car can have, for example, electric front windows and manual rear windows. In the feature diagrams graphical notation, the OR relationship is depicted as a black semicircle that connects the edges.
- **XOR.** As for the OR-containment, the XOR-containment relationship represents a containment between a parent feature and *a set* of children features. It means that the parent feature has to contain *only one* of the children features. In the Figure 2.1 example, the relationship between “Gearbox” and its children “Manual” and “Automatic” is a **XOR** containment because a gearbox cannot be automatic and manual at the same time. In the feature diagrams graphical notation, this relationship is depicted by means of a white semicircle connecting the edges.

Some clarifications need to be made when taking into account mandatory and optional feature categories in combination with the containment relationships. Although the XOR-containment between a parent feature and its children do not explicitly impose any constraint about the presence and the number of mandatory and optional contained features, the presence of more than one mandatory child feature has to be considered as an error. The XOR-containment, indeed, imposes the presence of *only one* of the contained features and, consequently, one or more of the remaining mandatory features cannot be selected and this is in conflict with their mandatory definition.

Without being considered as an error, the presence of exactly one mandatory feature in a XOR-containment does not make sense since its selection is obliged and no other of the eventually present optional features can be selected at the same time. For this reasons, in the feature diagrams presented

in this work, a XOR-containment relationship will never contain mandatory features.

On the other hand, the presence of more than one mandatory feature in an OR-containment relationship is perfectly legal since the OR-containment imposes the selection of *at least one* of the contained features, therefore, all the mandatory contained features will be selected and the other optional features remain still selectable. The same principle can be extended to the one-to-one containment relationship. Finally, when an OR-containment relationship involves only optional features, *at least one* of them will be selected, hence, the OR-containment relationship makes one of them virtually mandatory without specifying which.

An optional parent feature can have one or more mandatory child features. This is not in contrast with the previous considerations since it means that if, and only if, the parent feature is selected, the mandatory child features must be selected too. In contrast, if the parent feature is not selected, no one of its children can be selected, regardless of whether they are mandatory or optional.

The basic feature models define also two additional constraints between features, namely: *requires* and *excludes*. These two constraints allows the definitions of predicates regarding the contemporaneous presence of a set of features in a valid configuration of the modeled domain. They typically expressed in the form:

$$A \{requires|excludes\} B$$

where A and b can be simple features or a boolean composition of several features related by standard boolean operators (AND, OR, XOR, NOT). The presence of a feature has to be considered as a *true* value when evaluating the boolean composition of features.

- **Requires.** It means that if feature A is present in the particular configuration of the modeled domain, then, the feature B must be present as well. In general, if A and B are logical rules, the constraint imposes that if A is *true*, then, the rule B must be *true* as well.


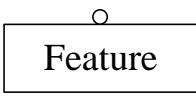
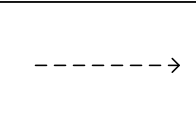
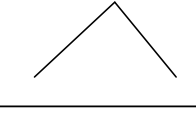
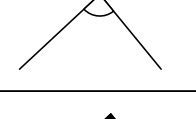
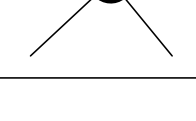
Notation	Description
	Mandatory Feature
	Optional Feature
	Reference to Feature
	One-to-one decomposition edges
	XOR-decomposition edges
	OR-decomposition edges

Figure 2.2: Feature modeling symbols.

- **Excludes.** It means that if feature A is present in the particular configuration of the domain, then the feature B cannot be present. If A and B are logical rules, the constraint imposes that if A is *true*, then, the rule B must be *false*.

Features of a feature diagram can be put into a generic relation with each other. This is useful to express references between features that cannot be expressed by the requires/excludes relationship, for example when there is the need to add some specific information to the feature model. In the feature diagram graphical notation, these connections are expressed by means of a dashed arrow connecting a couple of features and, optionally, with a short textual description of the reference on the arrow edge. Figure 2.2 shows and

summarizes the adopted graphical notation for feature diagrams.

2.3 Feature-based 4C Classification

This section presents the adopted classification strategy through the use of feature models, represented here by means of the feature diagrams. Essentially, a feature diagram is a hierarchy of common and variable features characterizing the set of instances of a concept in a given application domain, in our case the domain is represented by the component models to be classified and analyzed, the concepts are: the *Communication*, the *Computation*, the *Configuration* and the *Coordination*.

When adopting this approach for the analysis and the classification of a component model we aim to separate and classify the *areas of variation* that make a component model different from another one. Features can be selected according to the component model characteristics and respecting the composition rules of the feature models. The outcome is a “pruned” version of the diagram tree where the unselected features are omitted and the remaining features represent the concepts addressed by the component model under analysis. In this sense, the features provide a terminology and representation of the design choices, the available constructs, artifacts, instruments and means offered by the component models.

2.3.1 Separation of concerns

We identify the main independent areas of variations of a component model. This separation of concerns follows the separation of component design principles defined in [68]. Keeping these four aspects clearly separated is fundamental for facilitating the development of large maintainable and reusable software systems as it makes the definition and the implementation of components as independent as possible from the variability in hardware, in functionality and in application. Software components developed keeping in mind this separation of concerns are more reusable and more maintainable. As a matter of fact, the greatest part of the component models pursue such a separation

of concerns but, while having the same goal, they deeply differ in the manner and in the achieved results.

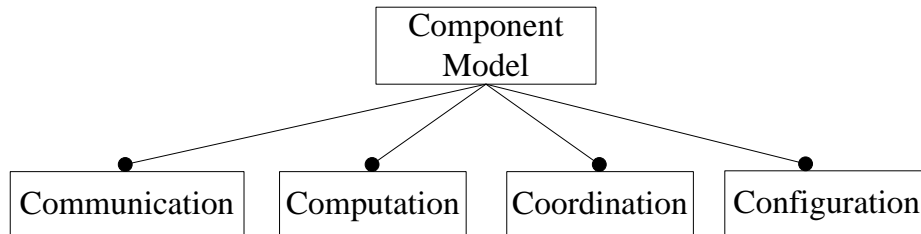


Figure 2.3: Top-level feature diagram for component models classification.

Figure 2.3 shows the top-level feature diagram with the five main areas of variation, namely the 4Cs key features, or concepts, of a component model. According to the main feature diagram a software component model is a definition of:

- **Communication:** expresses the interaction mechanisms and protocols for the exchange of data among components.
- **Computation:** represents the semantics of components, that is, what components are meant to be. It is concerned with the data processing algorithm required by the application.
- **Configuration:** expresses the composition of components making their assembly mechanisms explicit.
- **Coordination:** is the orchestration of the various system components to achieve a common goal.

Note that all top-level features are mandatory. Although this is clear for communication, computation and configuration features, the choice of presenting the coordination feature as mandatory needs for some clarification. Every component-based software system needs some kind of coordination mechanisms to make the components work in an harmonized way achieving the common goal but, as observed in several available component models and,

as stated in [68], coordination mechanisms are not always made explicit, not in the application point of view, nor at the component model definition level.

In many systems, the necessary coordination policy is hidden inside the component implementation, hence, it could be related to the computation concern. In this work we aim to keep the coordination concept clearly separated from the computation concern, even when analyzing component models that do not explicitly provide constructs, artifacts or means for coordination purpose. This approach lead to a better separation of concerns in the developed classification framework.

2.3.2 Communication

Communication deals with the exchange of data [68], in particular, the communication defines *how* components communicate with each other. Components need to communicate with each other in order to cooperate and component cooperation is the basis of any system, hence, the communication concern plays a fundamental role in defining both functional and non-functional characteristics of the entire component-based software system.

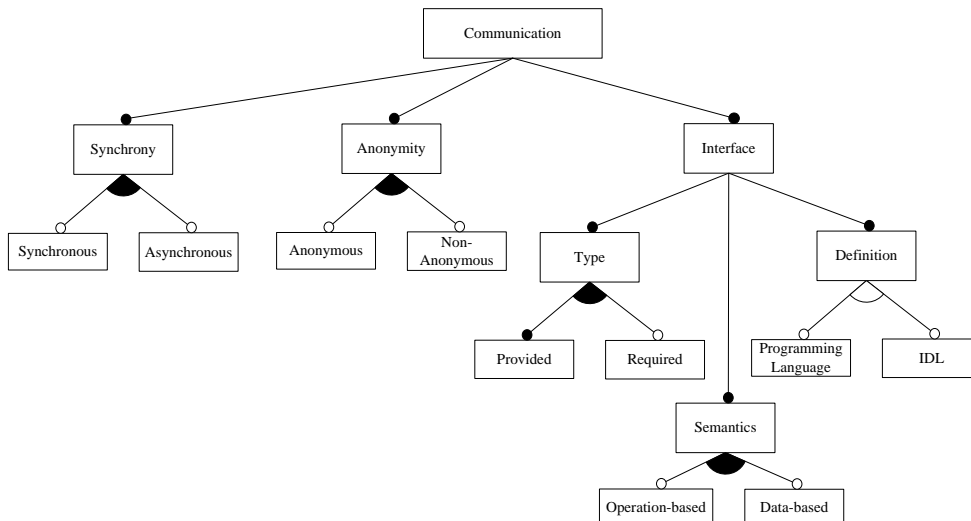


Figure 2.4: The Communication feature diagram.

The diagram shown in Figure 2.4 depicts the feature diagram of the communication concern. We can identify three main sub-features: *anonymity*, *synchrony* and *interface*. The anonymity feature is related with the *space decoupling* concept, the synchrony is related with the *synchronization decoupling* concept and the interface feature deals with the major construct provided by component models to administrate the communication among components.

Anonymity

Two communication mechanisms are deeply rooted in Object Oriented Programming, namely the caller/provider mechanism and the broadcaster/listener mechanism [48]. The former is involved when an object invokes another object method, the latter gives to the objects the capability of broadcasting and listening events or data. One the most notable interaction scheme involved in the broadcaster/listener mechanism is the publish/subscribe interaction thoroughly analyzed here [48].

Taking into account these two main mechanisms, caller/provider mechanisms usually led to an imperative communication style, as in the case of a command issued by one component to another one. On the other hand the broadcaster/listener mechanism usually led to a reactive communication style, as in the case a component reacting to an event notification by calling a callback function or method.

These two approaches play a fundamental role when taking into account the *anonymity* aspects of communication: when the communication between two or more components is *anonymous*, the communicating components do not need to know each other, i.e. the caller does not hold a reference to the provider, and the publisher does not know which subscribers are participating in the interaction [36]. In case of anonymous communication, the communicating components are defined to be “space decoupled”.

When the communication is *non-anonymous*, the communicating components need to have some knowledge about other components involved in the communication. For example, a caller component may need to hold a

reference to the provider component, or a publisher component needs to know which and how many interacting components are subscribed. Another kind of non-anonymous communication is concerned to the addressing mechanisms. In case of non-anonymous communication, the involve components are defined to be “space coupled”.

In some cases, the communicating components do not need to hold an explicit reference to the interlocutor but they still need to know some kind of address identifying the other components joining the communication i.e. the IP address and the port number for a socket-based communication. We can state that, although this kind of coupling is less tight than the previously introduced cases, the components are still coupled.

Space decoupling strongly increases the component reusability and the system flexibility because it removes explicit and implicit dependencies among components allowing for an easier replacement of components and for a more flexible reconfiguration of component interconnections.

Component models address the anonymity concept with several different approaches reaching different degrees of anonymity. Data-based communication style, usually implemented by means of data ports, allows a higher level of anonymity since each component do not need to know which component is sending data to its input ports and which component will receive data written on the output port. Exchanged data is stored into messages without address. The component may make assumptions about the inter-arrivals time of input data and about the type and format of the received messages but this knowledge is not concerned with the interlocutors’ specific instances. Port-based communication is offered by several component models such as OROCOS, RUBUS, AADL, Pecos and SaveCCM. In many cases the necessary port binding knowledge is managed outside the component definition. In OROCOS, for example, the component connections are specified in an XML-based deployment file where component ports are matched on the basis of their connections names. When adopting port-based communication, components can play the role of *data consumers*, *data producers* or both.

Another mechanism that allows anonymity is based on the publisher/-subscriber paradigm. Publisher components can publish messages on topics

without knowing explicitly which components are subscribed to these topics. On the other hand, subscribers components can subscribe, and hence receive messages, from topics without knowledge of the component that is publishing on the topic. The topic-based communication offered by the ROS component model is a clear example of this approach.

When the component interaction and communication is operation-based, for example when a caller/provider paradigm is adopted, components need to have some knowledge about the other components that are joining the communication and hence, the communication is non-anonymous. In these cases the component model can provide mechanism to reduce the knowledge needed by a components to carry on the communication. In the Service Component Architecture (SCA), a component can provide operations through the *service* construct and require operations through the *reference* construct. References and services are bound in such a way that a component can call the methods defined by a service through its reference. In this case the caller component just need to know the interface implemented by the provider since the binding with the current instance of the provider is managed by the SCA run-time system following the instructions defined in a SCDL-XML deployment file (composite file). This reduces the dependencies among components allowing for better decoupling.

Some component models provide both port-based and operation-based interface construction, for this reason the two features are connected by OR-decomposition edges in Figure 2.4.

Synchrony

Synchronization between communicating components is usually related with the caller/provider communication mechanism. The communication between two component is *synchronous* when the caller, after requiring a service to the provider by calling a function or a method, remains blocked waiting for the response. The execution time needed by the provider to produce the response can vary depending on the complexity of the computation, the load of the system and the communication delay introduced by the communication

mechanism. This kind of blocking communication mechanism is suitable when a response from the provider is needed immediately or within a certain, and generally short, period of time. The absolute value of the maximum allowed response time is an application-specific parameter.

On the contrary, *asynchronous* requests return immediately after the provider has received the parameters of the invoked operation and the result of the request can be retrieved later by the caller. It exists a slightly different version of the asynchronous request mechanism, the so called deferred-synchronous request that, when the response is ready, returns a token that allows the caller to retrieve the operation results when needed.

Component models usually provide means for remote operation call between components by exploiting communication mechanism supplied by the underlying middleware, for example the OMG Common Object Request Broker Architecture (CORBA) [6] or the Java Remote Method Invocation (RMI) framework. The CORBA framework is used in some component models such as OROCOS (in non hard real-time application) and in the Corba Component Model (CCM) itself. Both CORBA and RMI consist in making the remote operation calls appear as they were done in the caller's address space. This is carried out by means of a couple of tied components: the *stub* and the *skeleton*. The stub is a proxy of the provider in the caller's address space, it exposes the same interface of the remote provider. The skeleton resides in the provider's address space. So doing, the caller can invoke a stub method as it was in its local address space, then, the stub can serialize the request and transmit it on the network to the skeleton which will handle the request by calling the corresponding method of the provider. A similar couple of components is used to transmit the response back to the caller. This middleware-related mechanism allows a form of anonymity of communication since the caller and the provider do not need to know their mutual instances but only the stubs. Depending on the implementation of the provider, the stub-skeleton mechanism can offer support both for synchronous and asynchronous requests, for this reason the features are connected by OR-decomposition edges meaning that *at least* one feature must be selected (refer to Figure 2.4).

When the communication between components is data-based, it always

allows asynchronous communication since there is no reason for the data producer to wait for any kind of response from the data consumer, which may not exist. Component models and execution frameworks usually implement the data-based data exchange by means of ports that, in turn, are implemented by shared memory areas, the access to these areas is guarded by mutual exclusion mechanisms provided by the framework itself or by the underlying middleware in a way that it is completely transparent to the components implementation. The same concept can be extended to the topic-based communication provided by component models like ROS.

Interface

According to Medvidovic et al., a component’s interface is a set of interaction points between it and the external world [63]. Regardless of the particular features or of the scope of a component model, the interface concept is an obligatory part of the component specification, consequently, interfaces represent first-class entities in all component models, hence the interface feature is mandatory in the communication feature diagram shown in figure 2.4.

A component has one or more *provided interfaces* and/or one or more *required interfaces*. The provided interfaces of a component represent the contractually specified functionality that the component offers to their clients. The required interfaces, instead, specify functionality and resources that the component needs to perform its own functions and fulfill its own obligations to its clients. Not all component models offer an explicit notion of required interfaces, for this reason, in the feature diagram depicted in Figure 2.4 only the feature corresponding to the provided interface definition is mandatory.

Depending on the underlying semantics involved in the communication, interfaces can be classified into *operation-based interfaces* and *data-based interfaces* [35]. Operation-based interfaces are defined in terms of functions and parameters to denote the services provided or required by the component. Some other component models, instead, introduce the notion of communication port as interface element for sending and receiving, or providing and

requiring, data from and to the surrounding environment of the component with a data-oriented behaviour. The surrounding environment, in this case, can be considered as the set of the other cooperating and communicating components of the system. In many cases the existence of a data-based interface definition does not exclude the possibility of having operation-based interfaces in the same component model like in OROCOS.

Besides the port-based definition, some component models like ROS, provide another mechanism for data-oriented interface definition: the concept of topic. Topic-based communication is tightly related to the publisher/subscriber communication paradigm. Components can publish messages to topics that represent message container artifacts. Other components can subscribe to topics and being notified when a new message is available. Topics are defined by an identifier (URI), a message type and a set of publisher and subscribers components. This mechanism allows a many-to-many anonymous and asynchronous communication scheme.

Port-based communication and topic-based communication address the same goals related to the data-based semantics: making the communication between components anonymous and asynchronous. However, the data-based semantics for interface definition do not exclude the possibility of defining other interfaces with operation-based semantics and behaviour by the same component model. For this reason, *operation-based* and *port-based* features are connected by OR-decomposition edges meaning that *at least* one of the features must be selected.

Interfaces can be defined in two ways: by using the same possibly object-oriented programming language adopted for the component definition or by using an Interface Definition Language (IDL). Component models supporting multiple component implementation languages, like the Corba Component Model, or without a reference implementation language, e.g. Koala, usually adopt an IDL to define provided and required interfaces. By using an IDL, the interface definition is made independent from the component implementation allowing two different scenarios: first, the interface definition can be reused in various component implementations, second, the interface implementation can be provided by different component implementations allowing, for exam-

ple, the composition of systems with components implemented in different programming language.

When interfaces are defined using the same object-oriented implementation language, the semantics of the language can limit the possibility of defining required interfaces. It is the case of the Java language where the interface construct can be easily adopted for expressing provided interfaces but no construct is available for explicitly define the required interface concept. In these cases the programming language is often enriched by component model specific notation; this is the case of SCA where Java annotations are used to express clearly the presence of provided interfaces (services) and required interfaces (references). Anyhow, the use of an IDL usually excludes the possibility of using the programming language for the interface definition and vice versa. For this reason the corresponding features are connected by a XOR-decomposition edges in the diagram in Figure 2.4.

2.3.3 Computation

Computation is concerned with the data processing algorithms required by an application [68]. Data processing algorithms are defined in terms of: *data structures*, representing what is manipulated by the algorithm, *operations*, describing how data is transformed by the algorithm, and *behaviour*, describing the order in which operations are performed on the data. A Software component is a computation unit that encapsulates data structures and operations to manipulate them.

The diagram shown in Figure 2.5 depicts the feature diagram of the communication concern. Four main sub-features can be identified: the *component implementation language*, the *component behaviour*, the *real-time support* and the *distributed system support*. The component implementation language refers to the language offered by the component model to define the implementation of the components, the component behaviour is related to the presence of an internal predefined finite states machine managing the behaviour of the component, the real-time support feature is selected when the component model provides explicit instruments to manage the real-time

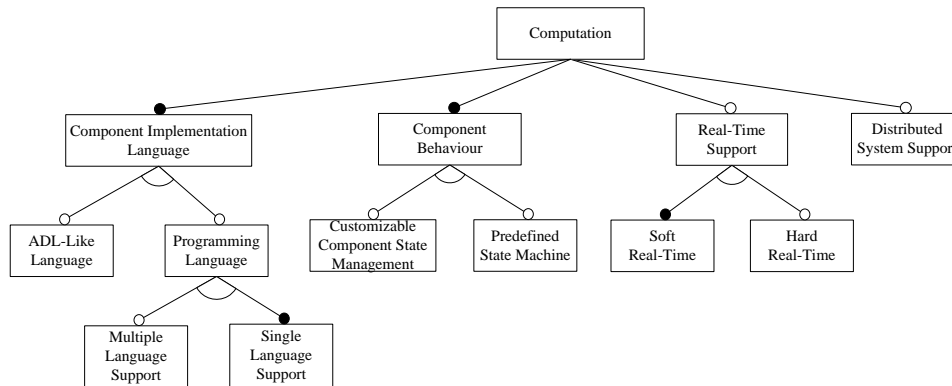


Figure 2.5: The Computation feature diagram.

execution of the components and the distributed system support is selected when the component model allows the execution of components in more than one unit of computation composing a distributed system.

Component Implementation Language

A Component is a unit of computation or a data store; i.e., a unit of computation and a state representation of the component. The definition of components requires a *component definition language* which may be distinct from the implementation language, and the associated run-time environment, used to actually implement components.

In object-based component models where components are defined in terms of classes, the definition language for components can coincide with the implementation language, for example, a component is defined as a class both in JavaBeans and in Enterprise Java Beans (EJB). Component models in which components are specified by using a programming language can be divided into two categories: models supporting one single reference programming language, like JavaBeans, and models supporting more than one programming language, like ROS and SCA. In ROS, for example, component can be defined in C++, Python, LISP or Java languages and they can cooperate in the same (distributed) system. The ROS core itself has been developed in these four

languages. The SCA component model is intrinsically multi-language capable since components can be developed in many languages like C++, Java and BPEL.

In some models, components are specified and implemented in a programming language and a specific IDL is provided to define the interfaces specifications. In these cases, the IDL is used to enrich the interface definition capabilities of the programming language and to make the definition of the interfaces independent from the programming language itself. The use of an IDL allows the definition of generic interfaces expressing the services offered by components; as explained in Section 2.3.2 this latter characteristic can be fundamental when multiple programming language are supported in order to facilitate the inter-operability among components implemented in different languages cooperating in the same component-based system, for example in Internet-based applications.

In general, if components are defined by using a specific programming language, at least one language must be supported for their definition. This trivial consideration is made explicit by the mandatory *Single Language Support* feature in the diagram in Figure 2.5. In the same diagram the single and multi-language support are connected by XOR-decomposition edges since the multi-language support includes the single language support.

In architecture-based component models, where components are architectural units, the definition language is an ADL [63] or a language with ADL-like features. In such models, the definition language is different from the (possibly zero or many) implementation languages. The UML modeling language is a notably example of such an approach since the same ADL-like graphical notation supports several implementation languages since it has been mapped on several platforms (like C++, Java, etc.). This can lead to the presence of more than one supported implementation language as in Fractal where a C implementation (Think) and a Java implementation (Julia) are available. OpenCOM provides similar multi-language implementation characteristics. In other ADL-based models there is no implementation language at all, this is the case of the Rubus component model.

Component models supporting the use of an ADL-like language to define

components do not allow the simultaneous use of a programming language for the same purpose, for this reason the two features are connected by XOR-decomposition edges.

Component Behaviour

The behaviour of a component is often regulated by the execution of a Finite State Machine, the distinction, in this case, regards the explicit definition of such a state machine. In the greatest part of component models there is no explicit notion of component state: components are activated as soon as they are deployed and then it is a designer task to organize the component execution.

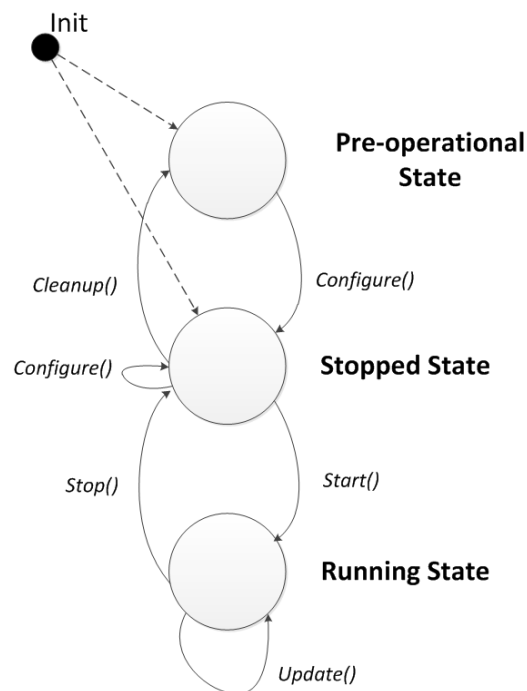


Figure 2.6: The OROCOS finite state machine.

In other component models, the execution of each component is strictly regulated by the presence of an explicit finite state machine which default states are specified by the component model itself. The most notable example

of such an approach is embodied by the OROCOS component model. For each component, the execution is organized into a three-state FSM as shown in Figure 2.6. In this case, the component behaviour is strongly influenced by the current machine state. Some operation are available only when the component is in a particular state, for example the component configuration can be done only in *pre-operational* state and, at the end of its operation, the component is switched in the *stopped* state. The execution of the function calls regulating the state switching is done by the OROCOS run-time environment (the so-called *deployer* component). The periodic execution provided by the OROCOS run-time environment is obtained, indeed, by periodically calling the *update* function in which the periodic task of the component is specified.

In other component models, the developer is free to provide his own state machine specification by hard coding it in the component implementation or using an external tool or library.

Some component models provide instruments to partially manipulate execution state of the components. In SCA, for example, Java annotations can be adopted to mark an initialization and a destroyer method that are called by the run-time environment (e.g. Apache Tuscany) at the component start-up and at the end of its execution. Other annotations can be used to specify the component initialization modality.

In general, when a FSM specification is provided, no other means to manipulate the component state are available, for this reason the features *customizable component state management* and *predefined state machine* are bound by XOR-decomposition edges in the feature diagram.

Real-time Support

This feature is related to the presence, in the component model, of specific means for managing the real-time behaviour of components. Real-time behaviours can be roughly divided into two main categories: *hard real-time* and *soft real-time*. In a real-time software system, the correctness of the computation depends not only on its logical correctness but also on the time in which it is performed. Real-time system must carry out tasks within a

certain deadline usually expressed in terms of absolute time (*absolute deadline*) or relative time referring to the release time of a task. Real-time systems are classified by the consequence of missing their deadlines: in *soft real-time* systems, the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service. In *hard real-time* systems the missing of a deadline is considered as a total system failure. Hard real-time systems are adopted when it is imperative to react to an event within a certain period of time or when the strict periodic execution of a task is the main goal. Examples of hard real-time systems medical devices such as pacemakers or closed-loop control systems where the periodic execution of the control algorithm is needed to achieve the control goal (e.g. regulating the shaft speed of an electric motor). If the periodic execution is not observed, the system can behave unexpectedly or can become unstable and result in damages. Soft real-time systems are used when the need for a strict periodic execution, or for a certain reaction time, is still important but the deadline missing is an acceptable event. Examples of these systems are live audio-video systems in which violations of timing constraints result in degrades quality, but the system can continue to operate.

When multitasking is involved, as in the greatest part of software systems, the execution have to be organized by a scheduler. The scheduling policy is normally priority-driven and, in a real-time system, the Early Deadline First (EDF) scheduler is often adopted. With this scheduling policy, the component with the earliest deadline is scheduled as the higher priority task to be executed at any time.

A component model supporting the real-time execution of software components must, at least, provide means to specify execution periods of periodic components. The component model and the underlying run-time execution environment usually relies on real-time operating system dependent constructs such as real-time timers, interrupt handlers and low-delay communication infrastructure to meet the component execution constraints. Some of the best known real-time operating systems are: LynxOS [5], QNX [10], RTLinux [13] and VxWorks [24]; alongside these, there are real-time extensions for Linux-based operating systems like RTAI [11] and Xenomai [25].

The real-time support is an optional feature for a component model as the computation feature diagram shows. Only a few component models support the real-time specification and execution of a component-based software system and, even when a certain level of real-time behaviour is supported, the possibility to support hard-real time constraints is very infrequent, even in those component models which are explicitly designed for embedded systems.

The Robot Operating System, although it does not provide any support for hard real-time execution, lets the user to specify component periodic execution and provides to the developers a low-delay topic-based communication that can be sufficient to accomplish robot control tasks with soft real-time constraints. On the other hand, OROCOS gives to the developer the possibility to specify periods and priority constraints for each component allowing for a hard real-time behaviour. It must be said that, in order to exploit the hard real-time features, the entire run-time core of OROCOS must be compiled and deployed on a real-time capable operating system like Linux with RTAI extensions or Xenomai.

Beside the already mentioned ROS and OROCOS, also Rubus and Pin provide means to define and manage real-time constraints for a component based system. However, the greatest part of component models do not provide this kind of instruments both because this is outside the scope of the component model, like in SCA, or because the real-time execution definition and capability is delegated to the developer that is in charge of exploiting the real-time capability of the underlying operating system.

Distributed System Support

Most of the component models support the execution of component-based software system on several networked computational nodes. The differences are related to the constructs, the instruments and the means that the component model provides to the developers for supporting the design and the creation of distributed component-based systems. Anyhow this feature is marked as optional in the feature diagram since the support for distributed systems goes beyond the base requirements of a component model.

Component models that refer to a specific middleware, like the Corba Component Model, exploit the distributed communication mechanisms provided by the underlying middleware to provide remote communication. In this cases the exchange of data with remote components is fully transparent to the components themselves since all the complexity is managed by the middleware.

Component models like ROS and OROCOS give the possibility to run the run-time environment on a particular node of the networked distributed system; this node which plays the role of the master. Knowing the URI of the master, all components can exchange data transparently as they were all executed on a local node. Although this mechanism is based on an underlying middleware (the non-hard real-time version of OROCOS, for example, uses the Corba request broker for communication) no knowledge about the underlying system or network configuration is needed by component implementations in this cases.

Other component models offer more advanced instruments to manage the communication between remote components. In SCA this is embodied by the concept of *binding*. A binding specifies the modality with which the communication must be carried out between two SCA components separating the component functionality specification from the communication specification improving the reusability of components. The binding mechanism also allows a SCA component to communicate with a non-SCA piece of software. When used to interconnect components executing on different units, the binding concept can be exploited to define the communication protocol and more than one binding can be associated to a single component allowing other components, or other software elements, to communicate through different modalities.

2.3.4 Configuration

Configuration determines which system components should exist and how they are interconnected [68]. The configuration of a component-based software system is defined by the component set, by the connection topology

between components and by the optional presence of connectors which may specify interactions and communication patterns. In many component models, connectors do not correspond to compilation units but manifest themselves in different ways such as shared variables, parameters, tables and so forth. In other component models, connectors are first-class entities and they are implemented as dedicated components. Connectors will be thoroughly analyzed in Section 2.3.5.

The component set and the connections between components define the *architectural configuration* or *topology* of the software system. The architectural configuration of a software system is often kept separated from the individual component specification since this separation of concerns makes the description, the comprehension and the manipulation of the entire system easier, both for humans and for machines [57]. In particular, the behaviour of a component-based software system depends on both the single components behaviour and on the connections among them. Hence, the configuration concern plays a fundamental role in the definition of the system: for this reason, the coordination has been identified as one of the four fundamental concepts for classification and analysis of component models. In Figure 2.7 we show the feature diagram of the configuration concern.

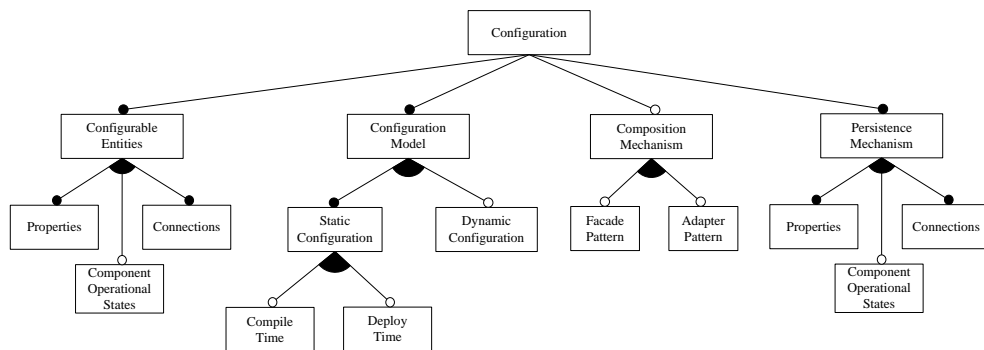


Figure 2.7: The Configuration feature diagram.

The analysis is divided into four main sub-features: the *configurable entities* that indicates which parts of the system specification can be configured; the

configuration model that defines the system development phase in which the configuration can, or have to, take place; and the *composition mechanism* that shows how components can be grouped in aggregated building blocks composing a system. The configurable entities and the configurable model are identified as mandatory features of all component models since they represent respectively the nature of the constructs that can be configured and the development phase in which the configuration takes place. On the contrary, the composition mechanism is a mandatory feature because, although it is a very common feature, not all component models provide instruments and means to group components.

Configurable entities

As already said, all component models allow some configuration mechanism but component models strongly differ when analysing the portion of the system configuration that can be directly configured or customized by the system developer.

The fundamental role of the configuration is the definition of the interconnections among components. If the adopted communication paradigm is data-based, the configuration defines the wiring between input and output ports, or the topics publishing and subscriptions, allowing the exchange of data between components. In component models where the adopted communication paradigm is service-based, the configuration defines the connection between required interfaces and provided interfaces. Since the collection of components and the interconnections among them are the mandatory characteristics for the definition of the component based system, the *connections* feature is marked as mandatory in the feature diagram; this means that, however the model is defined, it must provide the possibility to define inter-components connections. The specific instruments and tools made available by the component model for the configuration of the interconnections may vary. An XML file is very often used to configure the connections like in SCA, OROCOS and ROS. However, component models can offer multiple ways to configure a system, in OROCOS, for example, the XML deployment

file can be substituted by a configuration script written in the Lua scripting language; in ROS, components, called nodes, can express their intention to publish or subscribe on topics both by means of a XML-based launch file or by defining the topics names directly in the implementation code. Hard coding the configuration of component in the component’s implementation can seriously hamper the reusability and the flexibility of the system, for this reason, ROS launch files can be used to remap the hard-coded topic names.

Other component models directly support a graphical aided definition of components interconnections by providing graphical tools that assist the user in defining the system wiring. The support for graphical configuration tools is strongly enforced by the SCA architecture. The availability of an user friendly graphical tool is not only a comfort for the developer but can become a fundamental tool when large systems with tens of components need to be configured.

Components can have tunable parameters or properties that regulates their behaviour. The set of these parameters with their corresponding values and the specification of component connections, build up the system configuration. In all component models these properties can be externally set and, in many cases, this is done by using the same configuration file adopted for the connections specification. Since the parametrization of components properties greatly increase the components reusability and the system flexibility, all component models provide the possibility of specify properties, for this reason, the *property* feature is marked as mandatory.

The presence of a stored and retrievable configuration enhance the concept of *persistency* of the system configuration and greatly increases the ease of deployment, and re-deployment, of the system.

In component models where the behaviour of the components is regulated by a well defined state machine, it could be possible to specify the initial state of components. In OROCOS, for example, components can be configured to enter the “*start*” state as soon as they are deployed instead of waiting for an external initialization triggering event provided at run-time. The same effect can be achieved in SCA with a particular annotation that forces the early initialization of components. In SCA, indeed, components are by

default initialized by the run-time infrastructure as soon as another component requests one of the provided services for the first time. In some cases the initialization of a component can be a time-consuming operation since it may involve the establish of a remote connection, or the initialization of a database, and this lead to a delay in the first service call that can be unacceptable. To overcome this problem, the component initialization can be marked to be completed at the system start-up.

Configuration model

With the *Configuration model* we aim to identify the development phases during which the configuration of a system is possible. The life-cycle of a component can be divided, at least in first approximation, in three main phases: compilation, deployment and run-time. During the compilation phase the component code is translated into an executable entity that can be an application, a shared object, a dynamic link or static library, or a bytecode. Then the components are deployed: in this phase they are put into execution and linked together to form the system. At the end, in the run-time phase, components are actually executed and the system is operating. These three fundamental main phases can certainly be divided into sub-phases and further refined but, for the sake of this analysis, the simplified three-phases component life-cycle is sufficient.

The configuration of a component-based system can be carried out in each of these phases, in particular, if the component model offers means to configure components during the run-time phase, it allows the dynamic (re)configuration of the system. However, although the reconfigurability is allowed only by some component models, it cannot be classified as a mandatory feature. Moreover, the configuration phase can be carried out before the run-time phase in all the component models, for this reason, *static configuration* and *dynamic configuration* are connected by OR-decomposition edges.

The static configuration of a system can be done both at compile time, usually hard-coding the configuration instructions in the components code,

or at deploy time. The possibility of defining the configuration of a system before the compilation of components is an unusual feature and it is restricted to embedded system domain-specific component models, such as Koala, where the loss in system flexibility is compensated by a lightweight and run-time efficient management of the connections between components. In many cases the compile-time configuration is accompanied by the possibility of refining and optimizing the configuration at the deployment time.

The deploy time configuration represent the most broadly used strategy and it is usually implemented by interpreting the already mentioned configuration file. In many component models a special component is in charge of deploy and configure components on the basis of the configuration instructions. This is the case of the “*deployer*” component of OROCOS systems.

The run-time configuration can be performed by some, usually general purpose, component models. The mechanisms to manage the run-time configuration of a system vary from one model to another. In general the reconfigurability is managed by dynamically wiring components interfaces. This operation can be done on request, can be triggered by some external events or can be guided by the analysis of the system state in terms of load amount, quality of service, network congestion or other parameters. The dynamic configuration and reconfiguration is at the basis of the “quality of service” aware systems in which a certain level of service quality can be guaranteed by reconfiguring the system. With software qualities we refer to non-functional qualities such as efficiency, performance, completeness and exception handling [36]. The quality of the services provided by components depends on both their specific implementation and on the execution environment state. This latter is not known at deployment time, hence, instead of choosing components with reference to the worst case assumption, this selection can be made at run-time by the system itself through the analysis the total amount of system resource available. Knowing the characteristics of all the available implementations of the components, the system can select the most appropriate quality of service level for each component. For this, the presence of a reconfiguration mechanism provided by the component model is needed.

A less advanced reconfiguration mechanism is provided by those models which let the components to be added or removed at run-time. This is related to the run-time nature of each component. In ROS, for example, components, called nodes, can be deployed as stand-alone processes and hence they can be put into execution or turned off at run-time. This can be considered as a reconfiguration facility although there is no notion of quality of service.

Composition mechanism

Components can be grouped into higher level structures called, in general, assemblies. The configuration of component assemblies usually takes place at deployment time when binaries of components are loaded into the system. This enriches the architectural configuration of the system and let the developers to create groups of components that cooperate for providing a service. Component assemblies can be used, and also reused, together improving the modularity of the system.

Some component models do not provide means to group components and hence they impose a plain architectural configuration of the system where all components are considered as peers from an architectural point of view (OROCOS). Other models, instead, provide various means to group components, we can identify two main categories of composition mechanisms referring to the well known software design patterns: *facade pattern based* and *adapter pattern based* composition.

The *Adapter* is a structural design pattern used to convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [50]. This is done by developing a software wrapper that "adapts" the interface provided by a class to the interface that the class users (clients) expect. This can be achieved by simply converting data formats or by translating a single method call into a set of ordered calls and vice versa. When transposed to the component-based software domain, this approach can be realized by wrapping a preexisting component, or more often a set of components, into a container component. The container exposes a different interface to other components

of the system by intercepting the method calls and translating them into a set of calls suitable for the interfaces of the contained components. Hence, the container carries out some computation for method call ordering or for data type adaption.

As the Adapter, also the Facade is a structural design pattern. Its intent is to provide an unified interface to a set of interfaces in a subsystem. Facade defines an higher-level interface that makes the subsystem easier to use [50]. This is done by just delegating portions of the inner interfaces to the outside. When transposed to the component-based software domain, this approach can be realized by wrapping a set of components into a container component that exposes a subset of the contained components interfaces. This can simplify the overall interface of the set of components exposing only the operations that make sense to be seen from the outside.

The main difference between the adapter-like and the facade-like approaches is that the former adds some computation to the contained component set while the latter provides just a subset of the contained components interfaces. This difference is not negligible since an adapter-like container, or composite, is a fully fledged component with its own computation features while a facade-like container is only an architectural construct without its own computation. The composition mechanism can have impact on component and composite reusability. In OpenCOM, components can be composed into a semantically rich set of containers (capsule and caplets) that add composition and often coordination features to the component set. In SCA, components can be grouped into composites which are facade-like containers that expose a subset of the components interfaces to the outside without adding any kind of computation.

In the diagram shown in Figure 2.7, facade-like and adapter-like composition mechanisms are bound by OR-decomposition edges meaning that *at least* one of these two mechanism should be provided by component models that allows component composition. In some cases both of the mechanisms are provided since a container component can expose a subset of the internal components interface by promotion and, at the same time, adapt other parts of the components interfaces, for example regulating the access order

of methods. This is the case of the *container* concept in Corba Component Model.

Persistence mechanism

Each component model provides means to store and retrieve a particular configuration of components, we refer to this feature with the term *persistence*. Anyhow, component models strongly differ in tools and methodologies provided for this purpose as well as in which features can be part of a retrievable configuration.

The most adopted instrument for configuring and keeping trace of a system configuration is a *configuration file* or a set of files. Almost all component models allow the users to store a configuration in text file, usually XML-based, that defines the set of components and the properties values of each component, or refers to a specific property file associated to each component. This feature is present also in those component models in which components are standalone processes that can be manually executed. ROS belongs to this family of component models and, beside the possibility to manually execute single components, it allows the definition a set of configured and executable components by means of a launch file.

In this sense, an advanced feature is represented by the possibility of “freezing” the component operational state making the particular state of a component instance persistent in time. In this cases, component instances can be stopped during execution, stored as they are, and retrieved later. This feature is provided by the *Component Implementation Framework* (CIF) of the Corba Component Model.

2.3.5 Coordination

Coordination is concerned with the interaction of various system components [68]. It can be considered as the most complex feature to be analyzed in component model both because a great number of coordination mechanisms are possible among components and because the emerging behaviour of a component-based system is strongly influenced by the coordination of the

execution of its components.

Coordination is tightly coupled with interaction and concurrency concepts: components “cooperate” with each other locally or through a communication network to achieve a common goal and “compete” for the use of shared resources, computing capabilities and communication infrastructures [27]. In general, components are not insulated applications, instead, they share resources such as computational capabilities, network bandwidth, memory, access to shared resources like I/O lines, databases and so forth. Components also need to share results in order to achieve their common goals, the output of a component is often the input of another one and services provided by components are used by others. For these reasons, in *each* component-based software system there is the need to coordinate and orchestrate the execution of components.

Keeping the coordination concern separated from other concerns is a difficult task because, in many cases, the coordination between components is achieved by inserting the coordination mechanisms and policies directly into the component implementation (computation) or into the data exchange mechanisms (communication) or by properly configuring the components (configuration). Anyhow the separation of concerns between the coordination and the other features is one of the most important aspects to achieve successful reusability, modularity and flexibility characteristics in a component-based system. For this reason, classify component models on the coordination axis is convenient.

In Figure 2.8 the coordination feature diagram is depicted. The coordination concept is composed of two main sub-features: the *connector* notion and the *component roles*. With the term “connector” we refer to all forms of, implicit and explicit, inter-component connection mechanisms that include coordination policies. With “component roles”, instead, we refer to any kind of coordination policy offered by the component model to assign roles to different components.

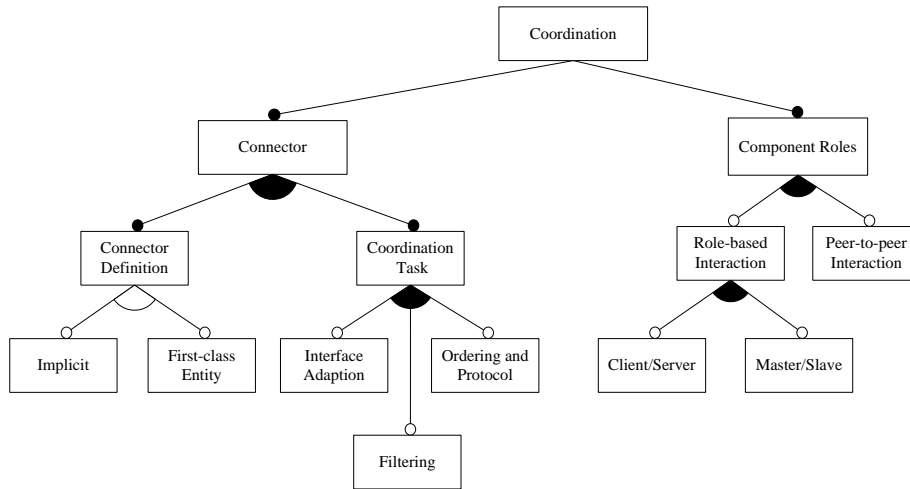


Figure 2.8: The Coordination feature diagram.

Connector

In most component models, connectors do not correspond to compilation units, but manifest themselves in different ways such as shared variables, separately configurable routing components, table entries, buffers, instructions to a linker, dynamic data structures, sequences of procedure calls embedded in code, initialization parameters, client-server protocols, pipes, SQL links between a database and an application, and so forth. This is a typical characteristic of the field of real-time and hard real-time applications, where software modules, or components, are usually developed as independent low-level pieces of software, for example kernel modules of a Linux-like system managing low-level hardware interfaces (drivers). In these cases the communication among components is managed by means of Inter Process Communication (IPC) techniques based on the synchronized and guarded access to shared memory areas such as FIFOs, message mailboxes, circular and swinging buffers as explained in [39]. In those cases, the connector concept is embodied in the adopted data exchange mechanisms, sometimes provided by the underlying operating system, and there is no specific construct or entity representing the connector concept.

On the contrary, in other component models, e.g. in [30] and [58], connectors are first-class entities modeled and implemented as special components, which specifically deal with interactions and dependencies among components.

Regardless if they are first-class entities or hidden inside component implementation, the connector concept is a mandatory feature of each component model and each connector must have a definition, either implicit or explicit.

The taxonomy of software connectors presented in [64] identifies several tasks that a connector can perform:

- **Interface adaption:** connectors can adapt interfaces by transforming the exchanged data between components or showing a modified view of a component interfaces to another one. This behaviour is almost the same of the composite explained in Section 2.3.4 with the difference that, in this case, components are not grouped into a composite objects and the interface adaption is directly done by the connector.
- **Filtering:** connectors can implement auxiliary computation on the data that is exchanged between communicating components, such as: cryptography, data compression, data segmentation and de-segmentation, incoming message filtering and forwarding, and so forth.
- **Ordering and protocol:** connectors can provide mechanisms for the coordination of requests, the ordering of method calls on component interface and the implementation of queues with various ordering policies.

The last item is particularly important for the coordination concern since the ordering of the method calls of a component interface is usually defined by its behavioral specification and it represent the contract between the component and its clients. A contract can be seen as an explicit roster of mutual obligations expressed in the form of various kinds of constraints, such as preconditions, postconditions, invariants, and protocol specifications. For example, a contract could specify that a method call must be invoked before another one to obtain correct results. This implies a form of coordination policy that, without an explicit connector, remains implicit in component implementation. With the use of connectors, instead, this can be clearly

specified allowing for an easier reuse of the components. Moreover, connectors are in charge of ensuring the compliance with the protocol of a set of interfaces by completely mediating the access to the component interface by the clients. In general, connectors represent the means to achieve the composition of individual software components into complex component systems.

Component Roles

Coordination is involved in the separation of roles between components composing a system. The definition of roles between components is a mandatory feature of all component models, even when all components are forced to be peers like in OROCOS or in ROS. In this kind of component models all components can produce data for other peers or consume data produced by other components without denoting an explicit separation of roles between service providers or petitioners. This is common when components are implemented as “filters” consuming data from input ports (or subscribed topics) and producing data to output ports (or published topics). In this cases the coordination among components is defined by the presence, or the absence, of data on the ports and the execution of a component may be triggered by the presence of new data on its input ports, independently by which component produced it. Similarly, the production of data can trigger the execution of one or more peer components receiving data from the component’s output ports.

A component model can make a clear distinction between components by defining roles. In service-oriented component models, like SCA, components can be servers that provide services to clients components or can be clients that use services provided by other servers or can behave as clients for some components and as servers with respect to others. This separation of roles let the developer of the system to define which component will require services and which other will provide them allowing for an explicit coordination policy.

In other component models, the execution of a component can be totally controlled by another component leading to a master/slave configuration. When such a kind of separation of roles is enforced, some components can

become *controllers* of others and then they can explicitly implement a coordination strategy. Usually, slave components actually carry out the functionality of the system and these pieces of functionality can be reusable in several different applications. At the same time, master component embody the control logic which is typically strongly application dependent. This separation between slaves and masters helps in clearly separating the application-independent specification from the application-dependent behaviour specification of the system and can make the reuse of single components, or entire parts of the system, much easier and effective.

Component Models Survey

A great number of component models have been presented and analyzed in literature. A certain number of them are specifically designed for particular applications, for example: Internet-based applications, embedded systems, robotic systems, database or data intensive systems and so on. Generally, these models try to address the application specific requirements at the component model definition level by introducing specific constructs or implicitly assuming the adoption of a particular programming language or a specific middleware.

Other component models are instead deliberately general purpose in the sense that their level of generality allows them to adapt to a wide range of different applications with different requirements. Generally, these component models do not impose a specific programming language or paradigm and they do not depend on a specific middleware or, at least, they can be supported by a widely used and well known middlewares, such as CORBA.

Hereafter, a set of component models is presented. For each of them a brief description is given. In Chapter 4 we will analyze five of them, namely: SCA, ROS, OROCOS, OpenCOM and Corba Component Model by applying the analysis and classification model presented in Chapter 2.

When listing (in alphabetical order) the component models we do not provide any product name or version number except for the cases in which the version number denote significant differences from previous versions of the same component model.

3.1 Corba Component Model (CCM)

Since 1989, the Object Management Group [6] has been standardizing an open middleware specification to support distributed applications. The CORBA Component model (CCM) [79] [78] [43] is an object-based component model evolved from OMG-CORBA Object Model. The CCM specifications define an abstract model, a programming model, a packaging model, a deployment model, an execution model and a metamodel. The metamodel defines the concepts and the relationship related to the models. Component are specified through an object-based component definition language which uses an IDL for interface description. Once components are fully defined, they can be implemented using the Component Implementation Framework (CID) which relies on the Component Implementation Definition Language (CIDL). The CID describes how component parts should interact with each other. The resulting component implementations can then be packaged into assembly files such as shared library or JAR files and linked dynamically. Finally, a CCM deployment mechanism is used to deploy the component in a CORBA component server that hosts the component implementation. Deployment and packaging instructions are specified by particular XML descriptors. In CCM, as in other object-based component modes, connectors are not first class entities and there is no explicit notion of architectural configuration or hierarchical composition mechanisms.

Just like a standard CORBA object, a CCM component instance is uniquely identified by an *equivalent interface*. Other components, or even component-unaware software, can invoke operation via a reference to a component's equivalent interface. As with regular CORBA objects, a CCM component's equivalent interface can inherit from other interfaces called *supported interfaces*. Each CCM component has a *home interface*. A client can access the home interface to control the life-cycle of each component instances it is currently using. This operation include the creation the destruction and the retrieval of instances.

A CCM component can interact with other CORBA artifacts, such as clients or other collaborating components, through *ports*. CCM specifies four

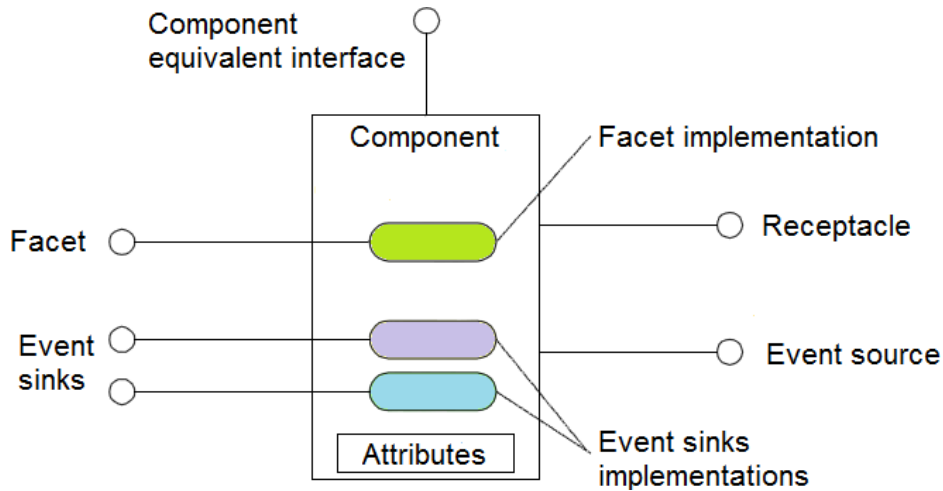


Figure 3.1: The Corba Component Model (CCM).

port mechanism, namely: *facets*, *receptacles*, *event sources and sinks* and *attributes*. Facets are provided interfaces that represent the functionality contractually provided by the component to its clients. A provided interface define operations that can be invoked synchronously by using CORBA two-way operations, or asynchronously by using CORBA asynchronous method invocation (AMI), by component clients.

A component receptacle represent a reference to other components instances. Using these receptacles, components can connect to other components or CORBA objects and invoke operations synchronously or asynchronously.

Event sources and sinks supports asynchronous event-based communication between components. A component declares its interest to publish or subscribe to events by specifying event sources or event sinks.

CCM extends the notion of attributes defined in CORBA. Components can expose attributes that can be set by a configuration tool during the system configuration, CCM allows operations that access component attributes to raise exceptions if an attempt to change a configuration is made after the system configuration has been completed. In figure 3.1 we show an example of CCM component with interfaces and attributes.

3.2 Fractal

Fractal [37] [49] is a hierarchical and reflective ADL-like component model developed by France Telecom R&D and INRIA. It is based on a XML-based ADL describing component types, implementations, hierarchies and bindings. The ADL provides an XML DTD that can be extended to integrate other user-defined architectural concerns. The use of a language independent Interface Definition Language (IDL) allows Fractal to be implemented in various programming languages; in particular, there are two reference implementations of the component model: *Julia*, which is written in Java, and *Think*, which is written in C.

A Fractal component is an encapsulated run-time entity described by Fractal ADL, each component is composed of a *membrane* and a *content*.

The *content* of a component consists of a finite set of other components called sub-components, each one of them can be composed of other sub-components creating a tree-style hierarchical structure. The *membrane* of a component collects all of its interfaces.

Interfaces are the only way of interaction among Fractal components. Interfaces can be of two kinds: *server interfaces*, which correspond to access point accepting incoming operation invocations, hence, they can be considered as provided interfaces; and *client interfaces*, which correspond to access point supporting outgoing operation invocations and can be considered as required interfaces.

As mentioned above, all component interfaces lie in the membrane of the component, in particular interfaces can be external or internal. External interfaces are accessible from outside the component while internal ones are accessible only from the sub-components.

In addition to the component specific interfaces, the component membrane may contain several *controllers objects* implementing *control interfaces*. Controllers are used to monitor and control the behaviour and the life-cycle of components by suspending, stopping and resuming the component itself or its sub-components. Controllers can also be used to dynamically configure or re-configure components implementing the reflective characteristics of Fractal.

Interceptor objects are a particular type of controllers that can intercept the incoming and outgoing operation invocations of an interface adding additional behaviour to the handling of such invocations. In figure 3.2 we show a typical Fractal component highlighting interfaces and sub-components.

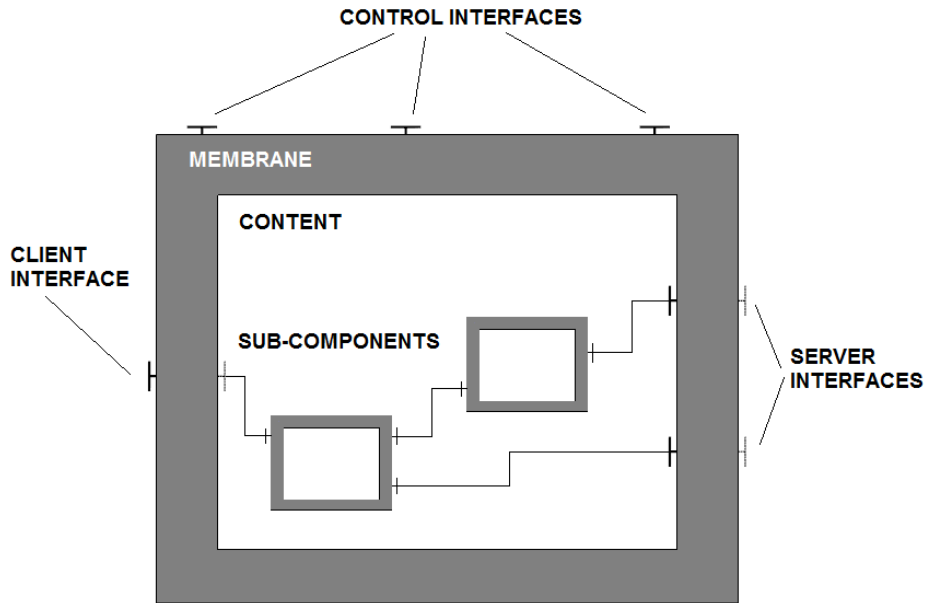


Figure 3.2: A generic Fractal component.

Communication between Fractal objects is only possible if their interfaces are bound through a *binding component*. A binding is a normal Fractal component whose role is to mediate communication between components, the model proposes two types of bindings: a *primitive binding* represent a binding between a client and a server interface that lie in the same address space, in this case the operation invocations emitted by the client interface are directly accepted by the bound server interface. A *composite binding* is a communication path between an arbitrary number of component interfaces composed of a set of primitive bindings and custom binding components. Composite bindings are used to build up complex synchronous and asynchronous communication scheme (stub/skeleton, client/server, publish/subscribe and others).

In Fractal there is no predefined set of bindings, except for primitive

bindings. Since bindings are Fractal components, composite bindings can be built by composition, just like other components.

All Fractal components can be composed of sub-components organized in a hierarchical tree structure where the parent-child relationship represents a composition relationship. A particular feature of Fractal component model is that a component can be included in several other components becoming a *shared component*. Shared components can be useful when representing shared resources like low-level system devices. More generally, shared components can avoid code duplication every time a single component has to be included in two or more containing components.

3.3 Koala

Koala [77] component model was developed and adopted by Philips for building software for consumer electronics like DVD players or TV systems. Koala has a set of modelling languages: Koala IDL is used to specify Koala components interfaces, Koala CDI (Component Definition Language) is adopted to define the components and Koala DDL (Data Definition Language) is used to specify local data of components. Koala supports C language for the implementation of components.

Koala components are architectural units of design, development and reuse. Components are able to communicate with the environment and with other components only through their interfaces. Components are defined by Koala CDI which supports both a textual and a graphical notation.

Components can be grouped into *compound components* in an hierarchical way. Compound components are defined with the same CDI used for the specification of single components.

Interfaces are the only mean of communication between components, in particular, all Koala interfaces are *operation-based* in the sense that they are defined by a group of functions and parameters. As in COM and Java, a Koala interface is a small set of semantically related functions. A component offers functionality to other components through its *provided interfaces*. Functionality can be required by the component through *required interfaces*.

Interfaces are defined using a simple IDL in which functionality are listed as prototypes in C syntax. In the component definition, each interface is labelled with two names. The first one is the *interface type name* that must be unique, the other one is the *instance name* and refers to the particular instance of the interface. This convention allows to have many interfaces of the same type on the border of a component as long as the instance name are different.

Koala supports both service interface and data interface, in particular service interfaces usually operate on the parameters passed as arguments while data interfaces expose state information of the related component.

In Koala components are designed independently from each other and can be linked through interfaces. When two interfaces are bound, their functions are connected on the basis of their name. Koala supports a static binding mechanism that is implemented using a simple tool also called Koala that, starting from components and interface descriptions, instantiates components and generates appropriate renaming macro in C header files.

Koala interfaces can also be bound using *modules* which are interface-less components that can be used to glue interfaces. Modules can also be used to implement basic components forming the leaves of the hierarchical decomposition of the model.

Koala also supports a limited form of dynamic binding achieved by the evaluation of a small subset of C expressions used to configure a particular type of component called *switch*. Switches are able to route function calls to different components on the basis of a conditional expression. If this expression can be evaluated at compile time, the dynamic connection may be turned into a normal function call without run-time overhead. More complex binding mechanism are implemented by *diversity interfaces* through which components can require information and properties describing their configuration.

As mentioned above, components can be grouped in an hierarchical way into compound components. A *configuration* is a set of components and compound components connected together to form a product. All required interfaces of a component must be bound to precisely one provided interface,

each provided interface can be bound to zero or more required interfaces. Developed components and compound components are collected into a global web-based repository for reuse purpose.

3.4 KobrA

KobrA (KOMPONENTENBASIERTE ANWENDUNGSENTWICKLUNG) [28] [29] is a hierarchical component model that supports UML-based representation of components developed by the Fraunhofer Institute for Experimental Software Engineering (IESE). Distinctive features of this model are the intensive use of UML diagrams for the representation of components and frameworks and the integrated support for software product line engineering and development.

KobrA components (often called “Komponents”) are architectural units defined with various UML diagrams capturing particular aspects of their structures, behaviour and functionality. No specific implementation language is proposed and the UML representation lets the developers choose the implementation language and the component technology that best fits the requirements.

The description of a Komponent is split into two main parts: the *specification*, which describes the externally visible characteristics of the component and thus defines the requirements which it is expected to meet, and the *realization* which describes how the Komponent satisfies this requirements. Component specification is described by four UML-based models (structural, behavioural, functional and decisional) while component realization is based on other four UML-based models (interaction, structural, activity and decision).

KobrA supports component reuse by replacing the realization of Komponenten with COTS or pre-existing components. Because of the hierarchical nature of the model, each KobrA component realizes its specification in terms of interactions with lower-level sub-components. Komponenten are organized into hierarchical tree-like structures called *frameworks* where each parent-child relationship represents a composition.

KobrA components are stored into a file-system repository which collects

implemented components and UML diagrams describing their features and textual documentation. They are composed by direct method calls in the design phase.

At the highest level of the KobrA hierarchy lies the concept of product-line, this means that all components related to a family of products are organized into a generic and reusable framework. This framework is then instantiated in a controlled way to provide specific software product as needed.

In contrast with most other approaches, a KobrA framework embodies all concrete variants of a family and not just the common parts. When the framework engineering phase is completed, specific applications can be instantiated from the framework selecting the needed features from a generic set. The instantiation of a framework is guided by a *decision model* describing the choices that distinguish distinct members of the product-line. The result is an application with the same structure of the framework but with all the unrequired features been removed. The application can then be transformed into an equivalent implementation that contains the instructions for automated compilation tools or can be manual translated into source code.

3.5 OpenCOM

OpenCOM [45] [44] is a lightweight ADL-like component model developed at Lancaster University targeting poor resourced systems like embedded systems. The component model uses a particular component definition language for the specification of components and OMG IDL for the description of interfaces. OpenCOM component model is language independent, a C and a Java reference implementations are available.

At the heart of the OpenCOM architecture there is a *component runtime kernel* that supports the services of loading and binding components. The kernel is able to support the run-time reconfigurability of components. For static systems, the kernel is used at deploy time to initially configure components and, after the configuration phase, it can be unloaded from the system in order to reduce computation and memory footprint. In dynamic systems, in which components need to be reconfigured or reloaded at run-time, the

kernel is kept running.

Above the kernel the *extensions* layer is present. Extensions are collected in two classes: *platform extensions* provide support for configuration of components at deployment time while *reflective extensions* provide support for dynamic inspection, reconfiguration and adaption of the component structure and behaviour at run-time.

An OpenCOM component is an encapsulated unit of functionality defined within an unit of scope and management named *capsule*. Components can be composed of other sub-components in a hierarchical way. The component model supports the notions of *caplets*, *loaders* and *binders* which are first class entities implemented as components. Caplets are nested sub-scopes within a capsule supporting encapsulation of components whose functionality are tightly-coupled. Loaders provides various ways for loading and unloading components into various types of caplets during run-time execution. Binders are used to bind interfaces and receptacles within a single caplet or across different caplets.

Components can support any number of *interfaces* and *receptacles*. An *interface* represent a functionality that a component offers to other client and it can be seen as a provided interface. A receptacle represent a functionality, or a resource, needed by the component to perform its specific task, it can be seen as a required interface. In OpenCOM there is no explicit distinction between data interfaces and service interfaces but all interfaces are defined in terms of functions and parameters in an operation-based way.

As mentioned above, the binding between receptacles and interfaces is managed by binders. The motivation for the use of binders is to represent different binding mechanism in the underlying system. For example, different binders can abstract over binding mechanism such as interrupts, buses, shared memory, remote method invocation and so on. Binders can vary widely in their complexity, especially when the binding is between components that reside in different caplets; in this cases a binder can include various communication mechanism like caller-provider or broadcaster-listener.

As well as in capsules and caplets, OpenCOM components can be organized into *component frameworks* which are a tightly-coupled sets of components

that can cooperate to address some specific area of concern. A component framework is implemented as a composite that can accept additional “plug-in” components that can increase or modify the behaviour of the whole composite.

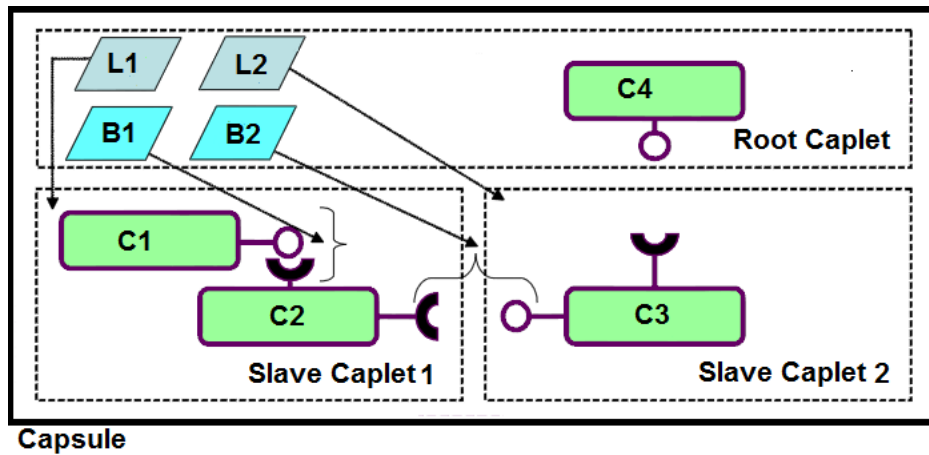


Figure 3.3: An OpenCOM capsule example.

Figure 3.3 shows three caplets within a capsule. Loaders (L1 and L2) and binders (B1 and B2) are defined in the root caplet; binder B1 manages the binding between components C1 and C2 in the first slave caplet while B2 manages the cross-caplet binding between C2 and C3. Loaders L1 and L2 allow the root caplet to load components of other caplets within the same capsule.

3.6 OROCOS

The Open Robot Control Software (OROCOS) [18] is a component-based, distributed and configurable software framework specific for the development of hard real-time systems e.g. data acquisition, signal processing, PLC functionality, waveform generation and robot motion control. The project was started at the University of Leuven (Belgium) in 2001 [38] and became usable in research environment in 2003 when a first version of a complete real-time motion control core for industrial robots was developed [40].

The emphasis of the project has always been on: flexibility, separation between software *structure* and *functionality*, distributed architecture of the control and hard real-time performances. The OROCOS framework is portable over several real-time and non real-time operating systems. In particular: Linux, RTAI, RTLinux and Xenomai.

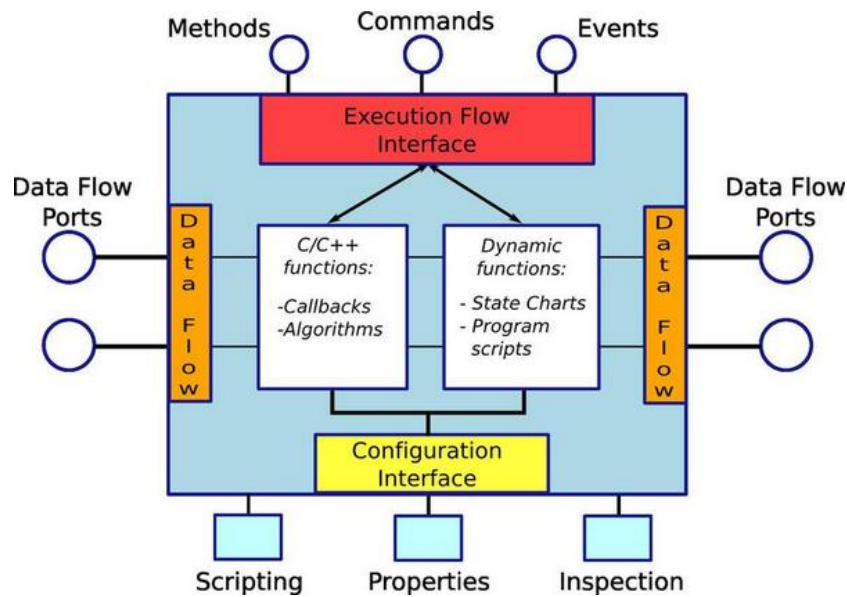


Figure 3.4: The structure of an OROCOS component.

The framework is based on a component model supporting loose coupling between components, separation between structure and functionality and event-driven interaction. An OROCOS system is composed of several *task context* representing software components implemented in C/C++ programming language. Each component, as depicted in Figure 3.4, is composed of:

- one **activity**: it is the component task and it defines its execution frequency, that can be null for non-periodic components, and its execution priority.
- n **ports**: The data exchange is based on the *port* artifact. Ports allow the flow of data among components from the output port of a component

to the input port of another one. Input ports can be defined as event port allowing the event-based communication paradigm.

- **m operations:** represent the functionality offered by the component to its peers. An operation can be considered as a provided interface and it is implemented as a method callable by other components.
- **j properties:** are parameters that allow the configuration of the component.
- **k peers:** represent the set of the other component of the system that offer operations that the current component can invoke.

The system is configured by declaring all the components and their respective properties, the ports connections, eventually defining a *connection policy* that regulates the data exchange, and the peers. The configuration is managed by a special component called *deployer* that configures the system following the instructions loaded from an XML file or by following a scripted procedure.

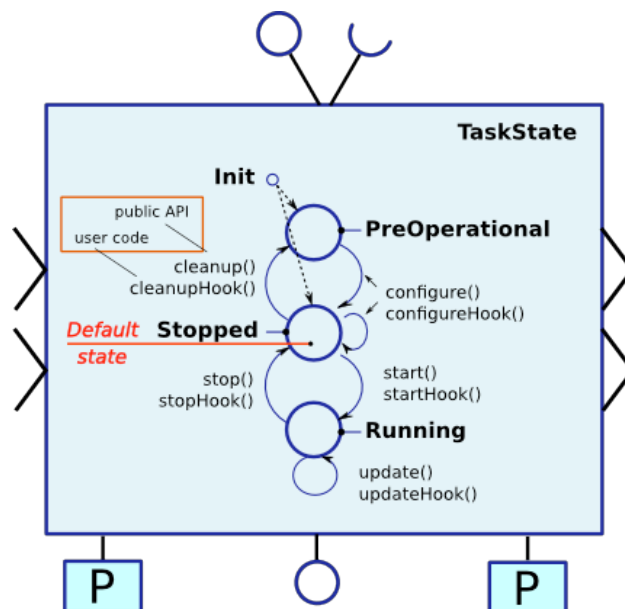


Figure 3.5: The internal FSM of an OROCOS component.

The execution of each component strictly follows an internal Finite State Machine (FSM) represented in Figure 3.5. At the startup the component is in the *PreOperational* state and it is ready to receive the configuration instructions. Once it has been configured (*configure()* or *configureHook()* calls trigger state transition) the component enters the *Stopped* state in which it can be re-configured or it can be put into the *Running* operational state by the *start()* or the *startHook()* calls. Once the component reaches the *Running* state, it will execute the defined task defined in the *update()* or in the *updateHook()* methods. To achieve a periodic behaviour, the deployer will periodically call the update method according to the execution frequency and the priority of the component. The *stop* or *stopHook* methods allows the component to return to the *Stopped* state and the *cleanup* or the *cleanupHook* methods make the component return to the *PreOperational* state cleaning the current configuration.

Being strongly oriented to the embedded systems and robotics domains, the OROCOS framework provides software libraries for the FSM definition and execution (rFSM library), for the bayesian filtering (BFL Library) and for the kinematic and dynamic chains computation (KDL library).

The OROCOS framework is distributed as a standalone framework under the LGPL license or as a part of the ROS environment. In particular, a ROS-OROCOS integration tool has been developed allowing the realization of mixed systems where OROCOS components can share data with ROS topics using ROS messages.

3.7 Orca

Orca [33] [62] is an open-source project developed by the Australian Centre for Field Robotics in the University of Sydney. It aims to apply component-based software engineering to robotics, focusing on application software development for mobile robots. Orca framework imposes as few constraints as possible to system developers who are free to design components that can provide or require any set of interfaces, implement those interfaces in any way they choose and compose a system from any set of components without architecture

constraints except from interfaces concordance between components.

Orca Framework must rely on a middleware addressing issues such as implementing application layer communication protocols and provide means for inter-component communication. First releases of Orca were essentially CORBA-based while newer versions are based on the Internet Communication Environment (ICE).

Orca Component Framework provides an ADL-like component model which defines interaction and composition standard for components. Interfaces descriptions and interactions are defined by a single XML configuration file per component. The component model follows the Orca's principles imposing as few constraints as possible to component definition and interaction.

In Orca, a component represents the implementations of algorithms and services for various types of hardware. To better understand the definition of component, *object* and *communication pattern* concept must be presented. An Orca object is an abstract definition of data that can be passed between components; Orca communication patterns are abstract policies describing how objects are sent between components. Consequently, a component can be implemented with knowledge only of objects and communication patterns. As an example, Figure 3.6 shows the *ServerPush* communication pattern: a single server can push objects asynchronously to many clients, in this case there must be a *single* Sender and *n* Receivers. In this context the term "client-server" implies only that the connection is one-to-many. Orca provides an on-line component repository of reusable components distributed under GNU-GPL licence.

Individual Orca components are free to provide or require any set of valid interfaces. There is no particular interface which all components must provide or require. However, some predefined provided and required interfaces are available and may be useful to access middleware-related features even if their use is optional. All Orca interfaces are operation based and developers are free to provide or require both data and service interfaces. All component interfaces are described in a particular section of the XML configuration file associated with the component.

A *transport mechanism* represent the method used to physically trans-

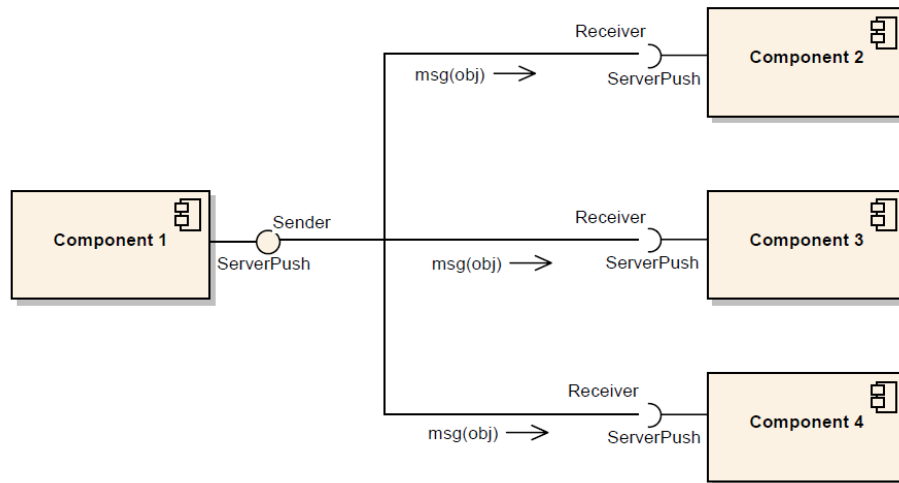


Figure 3.6: An example of the Orca ServerPush communication pattern.

port the data objects between the components composing an Orca system. Examples of those transport mechanism could be raw TCP/IP protocols, CORBA-based communication features for first releases of Orca or ICE - IceStorm mechanism used to implement event based communication in earlier Orca versions. Each component configuration file has a section for each interface describing: the binding between the current component interface and other interfaces, the transport mechanism used to implement the exchange of data and any necessary parameters needed to establish the connection. As already said, component implementations are transport mechanism independent, changing the transport mechanism between two or more components may require to add to or change the configuration file.

Orca does not specify any system architecture and so any architecture style defined by the system developers should be implementable. Because of this choice, there is no special component which is required to be implemented by all systems. Orca framework provides a set of optional predefined components addressing middleware-related features. Orca do not enforce the use of particular design patterns for either system or components but guidelines and working code for well-working designs are available on Orca on-line repository. The set of all XML configuration files of all components composing a systems

represent the global architecture configuration of the system.

3.8 Pin

The Pin Component Technology [65] [54] has been developed by the Carnegie Mellon University for use in Prediction Enabled Component Technology (PECTs). The core of Pin is a basic and simple component model for developing time and safety critical software for embedded systems.

Pin components are defined in an ADL-like language named CCL (Construction and Composition Language). UML Component Diagrams can be used to achieve a graphical notation of Pin Components while their run-time behaviour is represented with UML statecharts. Components are implemented in C language and distributed as dynamic link libraries (DLL). A component run-time environment provides services to access the underlying platform; for example file systems, timers, interrupts and I/O devices. The entire environment is based on a commercial real-time operating system or a standard operating system with real-time extensions.

A Pin component consists of two parts: an user-supplied custom piece of code and a Pin-supplied container. The interface structure of a Pin component is displayed in Figure 3.7.

Pin implements the container idiom for software components. Containers wrap the custom code and all interactions with the environment and other components are mediated by the container which may impose specific coordination policies. Referring to Figure 3.7, the custom code must provide an interface (*user code API*) that is used by the container to invoke, through the *user code plug-in* required interface, user supplied code in response to requests coming from other components or run-time environment. The container itself provides an interface (*container API*) used by custom code to request services from the run-time environment or other components. The container also provides a single interface to the environment (*Component API*) and two required interfaces (not depicted).

Each Pin component is independently implemented as a distributable dynamic library (DLL). Since components are totally encapsulated, all envi-

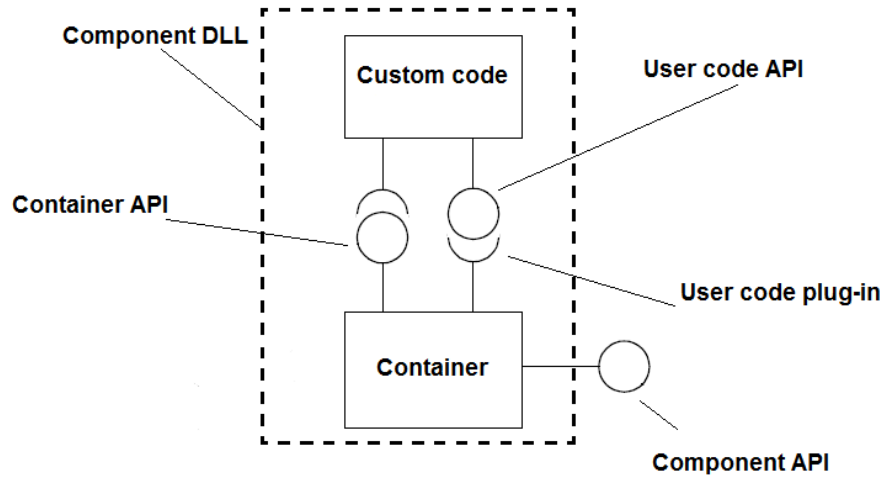


Figure 3.7: Pin component overview.

ronmental dependencies are fully explicit.

The custom code of a Pin component is organized as a set of one or more *reactions*, each one of them is executed in a dedicated thread created by the container. Custom code can have one or more *sink pins* and zero or more *source pins*. The execution thread waits for the arrival of a *stimulus* provided by a source pin, when the stimulus is received, the container executes the user code containing the reaction associated to the particular stimulus, at the end of the process the control is returned to the container which then checks for further stimuli and so on. During the execution, each reaction can generate a certain number of stimuli propagated to other components by source pins.

Pin supports synchronous and asynchronous interaction model. The former is implemented using a procedure-call mechanism, the latter has an event-based semantics.

Pin applications are constructed by connecting components using a repertoire of predefined connectors. Each connector can link a source pin to a sink pin. Connectors may impose coordination policies beyond those provided by containers, for example a FIFO policy on stimuli received on a sink pin. Pin supports a model of “pure” assembly in the sense that custom interaction code (glue code) is not permitted although the same effect can be achieved

encapsulating the interaction code in a new Pin component.

3.9 Pecos

The PECOS Component Model [80] have been carried within the Pecos Project (PErvasive COmponent Systems) with the support of ABB and academic partners. The project aims to enable component-based software development for embedded systems, specifically for field devices. PECOS is based upon an ADL called CoCo. The language is intended to be used for the specification of components, entire field device applications and system families. PECOS supports the formalization of the execution model of the components with Petri nets and their implementation with C++ or Java language. The behaviour of a PECOS component is not specified in CoCo but it is directly implemented in the target language.

In Pecos a component is an unit of design with a specification and an implementation. Each component has a name, a number of property bundles and a set of ports. The model expresses three types of components: *active components* which are characterized by having their own thread of control, *passive components* that do not have a thread of control and are used to encapsulate services that can be executed synchronously and *event component* whose functionality is triggered by an event. All of three kinds of components are characterized by properties encoding functional and non-functional information like timing requirements and memory usage. All components can be composed into *composite components*, hence, the whole application is often represented as an unique composite component. The behaviour of a component is a function or algorithm that takes, as input, data available on the component ports and produces results on output ports after computation. Since PECOS is used for specifying and developing embedded systems, each component usually represents a field device (an actuator, a controller or a sensor). A complete system typically represents a composite device running in a single periodic control loop. The execution is controlled by a schedule which defines the behaviour and the execution periods of the components.

PECOS is totally data flow oriented and ports are the only means of

interaction between components and their environment or other components. Because of this, all PECOS ports are data oriented and they are specified by: a unique name within the component, the type and the range of values of data passed over the port and the direction of the communication that can be *in* or *out*. According to this distinction, input ports can be seen as required interfaces since they collect data needed by the component for the execution of its specific function. At the end of the execution, the results are provided by output ports that can be considered as provided interfaces.

A port can only be connected to another port having the same type, compatible range and complementary direction through the use of a connector. Connectors can be implemented as shared instance variables describing the data-sharing relationship between ports. Ports of components can only be connected if they belong to the same parent component (composite), so, connectors may not cross the component boundaries.

As mentioned above, components can be grouped into composites in an hierarchical way. To increase the possibility of reuse of components, PECOS provides the concept of *abstract component* to specify families of components or architectural styles. This kind of component identifies a template component. An abstract component can specify *roles* which are placeholders for concrete instances. Abstract components and roles become concrete during the implementation of the component when they are filled in with the implementation of a specific component instance.

3.10 Robot Operating System

The Robot Operating System (ROS) [66] [20] is a free and open-source (BSD licence) robotic specific framework developed by Willow Garage ¹. ROS is not an operating system in the traditional sense since it does not manage the execution and scheduling of processes, rather, it provides a structured communication layer *above* the host operating system which is typically a Linux distribution.

¹<http://pr.willowgarage.com>

ROS is designed to be language-neutral and it currently supports five programming languages: C++, Python, Octave, LISP and Java. The ROS run-time manager (*roscore*) is developed natively in each supported programming language. The ROS specification is located at the communication layer and gives to the developers a set of instruments for the exchange of messages.

The entire ROS framework is based on a component model which is presented in current section and then deeply analyzed in Chapters 4 and in Section 5.2. The component model defines *nodes*, *packages*, *messages*, *topics* and *services*. Nodes represent the software components and they are deployed as independent processes running on a local machine or on a set of networked machines. The full set of cooperative nodes running on the network at a given time represent a ROS system.

Nodes communicates with each others' by passing messages. In ROS a message is a strictly typed data structure defined by means of a language-independent IDL. Custom messages can be created starting from primitive data types supported by the framework (integer, floating points, boolean, arrays, etc.) and then composed by nesting them at an arbitrary deep. Messages defined with the IDL are translated into serializable data structures at compile time.

The Robot Operating System supports both asynchronous and synchronous communication between nodes. The former paradigm is supported by the concept of topic while the latter is supported by the concept of service.

Nodes can send messages by publishing them to a given topic which is represented, at the developer point of view, as a string expressing the topic name. A node that is interested in a certain kind of data can subscribe to the appropriate topic. In general, there can be any number of publisher and subscriber nodes for each topic and nodes are not aware of each other' existence. When a new message is written on a topic, a callback function is asynchronously called in each subscriber node allowing it to fetch the new message.

ROS provides the service concept for supporting the synchronous communication between nodes. A service is defined by an unique string name (URI) and a couple of typed messages representing the service request and

the service response.

Nodes can be grouped into packages that allow the developer to compile, run and debug an entire set of components at the same time without adding any computational capability to the set. Packages are also used to wrap external libraries and they represent the minimal compilable unit of ROS. Packages are represented by a XML file containing the description of the interacting nodes and topics. At the top level of the hierarchy, a cluster of packages implementing a particular functionality can be grouped and distributed into a *stack*.

The Robot Operating System is a robotic application specific middleware, hence, it is enriched by a set of small yet useful tools for running, stopping, debugging nodes, for logging and visualizing message data, for managing geometric 3D transformation between frames and for automate compilation and link phases.

3.11 Rubus

The Rubus component model [52] was developed by Mälardalen University and industrial partners. It is a part of the Rubus Project and it was first introduced in 1996 for industrial purpose. The Rubus component model is intended for development of software architectures expressing data-flow and synchronization between software entities in single or multi-node systems. The component model is intentionally simple but can give enough expressiveness for development and analysis of resource constrained systems with a mix of hard and soft real-time requirements. The Rubus component model does not specify any particular language for the implementation of components, however the C language should be considered as the reference language for this kind of models.

Software Circuits (SWC) are the basic units of hierarchical decomposition in Rubus. SWC are defined by an ADL-based graphical component definition language, each one of them is characterized by its own behaviour, its interfaces and its internal state data. Interfaces are ACME-like ports that manage interactions between software circuits supporting the exchange of

data and the control flow among them. Software circuits can be composed into aggregate components named *assemblies* and *composites*. A certain number of assemblies, composites and software circuits form a *system* which is the top level hierarchical entity.

Output Data Ports and Output Trigger Ports can be conditioned or unconditioned, they can be considered as provided interfaces since they represent the functionality that the software circuits offer to their client. Input data ports and input trigger ports can be considered as required interfaces since they specify functionality, resources and data flow that the software circuit need to perform its own tasks. Interfaces can also be divided into other two orthogonal main categories: data interfaces and service interfaces. Input and output data ports belong to the former category since they provide a way to set or retrieve a property of a software circuit interacting with its internal state. Trigger Ports are instead a particular type of service interfaces that allow the control flow among the SWCs. Using C language as implementation language, interfaces are expressed as C header files containing the prototypes for functions provided by components.

Connectors are architectural building blocks that model interaction between components, in Rubus several items are used for this purpose. We can divide Rubus connectors into two main categories according to services they provide. Communication connectors support the exchange of data among the software circuits providing shared variables and initialization parameters. Examples of these connectors are the *source items* (input data ports), the *sink items* (terminator data port) and *named data items*. Coordination connectors deal with the synchronization of the control flow, in Rubus this aspect is embodied by the trigger signals. Trigger items like *clocks*, *interrupt generators* and *event generators* provide the support for the synchronization aspects, control flow items like the *down sampling block* and the *precedence block* are used to manage precedence and concurrency aspects regarding the execution of the software circuits.

3.12 SaveComp Component Model

The SaveComp Component Model (SaveCCM) [26] formalises the SaveComp Component Technology (SaveCCT) concept. Both SaveCCT and SaveCCM has been carried within the SAVE project. The goal of this project is the establishing of an engineering discipline for the systematic deployment of component-based software for embedded systems like vehicular systems. SaveCCT provides a component technology for the component-based development of software with support for accurate functional analysis and verification of safety critical software systems. SaveCCM provides a component model defining how component can be combined to create systems. According to the SAVE project requirements, this model supports the development of resource-efficient systems with a predictable behaviour.

The syntax of SaveCCM is based on a textual XML syntax. A modified subset of UML2 component diagrams is used as a graphical component definition language. The semantics is formally defined by a language called SaveCCM-Core and the transformed into timed automata with tasks.

In SaveCCM, components are built from interconnected elements called Components with port interfaces. The functionality of each component is typically provided by a single C function but the model allows the use of more complex components consisting of a number of communicating tasks.

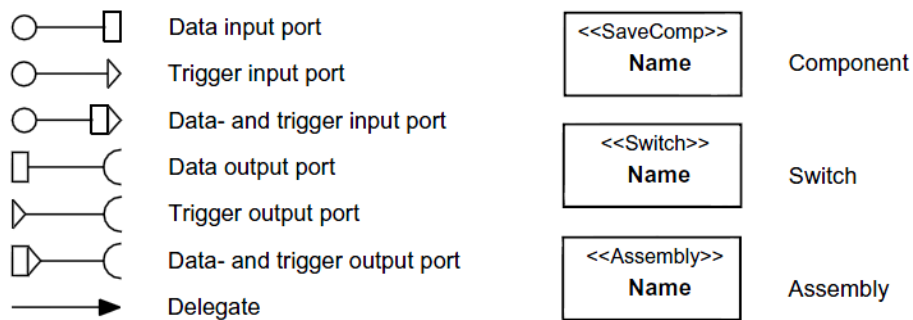


Figure 3.8: SaveCCM features summary.

The model is based on the “pipes and filters” paradigm with a rigid

distinction between data transfer and control flow. The former is captured by data input and output ports, the latter by trigger input and output ports. Components follows a “read-execute-write” semantics; at the beginning of the execution a component is inactive, it remains in this state until all input triggers ports have been activated. Once it is activated, the component reads all input data ports and performs the associated computations on the basis of the input values and its internal state. When the computing phase is over the output is written on the output ports and the trigger is propagated through the output trigger port, after that the component returns to the idle state. This strict semantics ensures that once a component is triggered, the execution is independent from any other concurrent activity in a non-preemptive way. The “read-execute-write” semantics facilitates the component functional analysis according to the SaveCCT goal.

Input data and input trigger ports can be viewed as required interfaces that specify functionality and resources needed by the component to perform its own functionality. Output data and output trigger ports are provided interfaces that represent the functionality made available by the component to its client. SaveCCM also offers an aggregate type of ports called “data and trigger port” that can be both input or output. In this case the data values and the trigger signals are available at the same time on the interface of the component.

SaveCCM ports can be divided into other two families: data ports are data interfaces that expose state information of components and make it available to the clients. Trigger ports are instead service interfaces that regulate the control flow among the components.

In SaveCCM, each port can offer two types of connections. *immediate connections* represent atomic loss-less transfer of data or trigger signals from an output port to an input port. This is the typical mechanism of interaction between two components located on the same physical node. For distributed systems, like vehicular ones, SaveCCM offers a more sophisticated connection concept provided by the so-called *complex connections* that represent data or control transfer over a real channel characterized by possible delay or information loss. In this cases the communication mechanism is modelled

with a finite state automaton describing the behaviour of the connection.

The model offers a particular type of connector construct named *switch*. Switches provide the means to change the component interconnection structure either statically (before compilation) or dynamically (at compile-time). Each switch specifies a number of connection patterns managed by logical expression over the data available at the input ports of the switch. Switches can operate on data ports or on trigger ports and can use partial evaluation to identify the part of the switch that will not change during run-time. Such static parts are optimized into normal connections and the components that are unreachable are omitted in the final system. Switches are not triggered by any signal, they respond directly to the arrival of data or trigger signals shortening or opening connections according to the evaluated logical expressions.

Components can be encapsulated into sub-systems called *assemblies*. The internal components and interconnections of an assembly are hidden from the rest of the system and can be accessed only in an indirect way through the ports of the assembly. As in UML2, a connection from an assembly I/O port to an internal component I/O port is denoted by a delegation arrow even if they are semantically the same as ordinary connections. Since assemblies do not provide any additional functionality, they should be seen as a mean for obtaining a collection of components hiding their internal state rather than a component composition mechanism. A number of components and assemblies wired together form a system that is the top level entity of a SaveCCM system as shown in Figure 3.9.

3.13 Service Component Architecture

The Service Component Architecture (SCA) [14] provides a set of specifications for the construction of distributed applications following the principles of the Component Based Software Engineering and the Service Oriented Computing (SOC). The SCA model was promoted by a group of enterprises operating in the field of the Information Technology such as IMP, SAP, Oracle and SUN. The first release of the SCA specification was published in 2005 and then refined in 2006 and 2007. The SCA specification will be deeply analyzed in

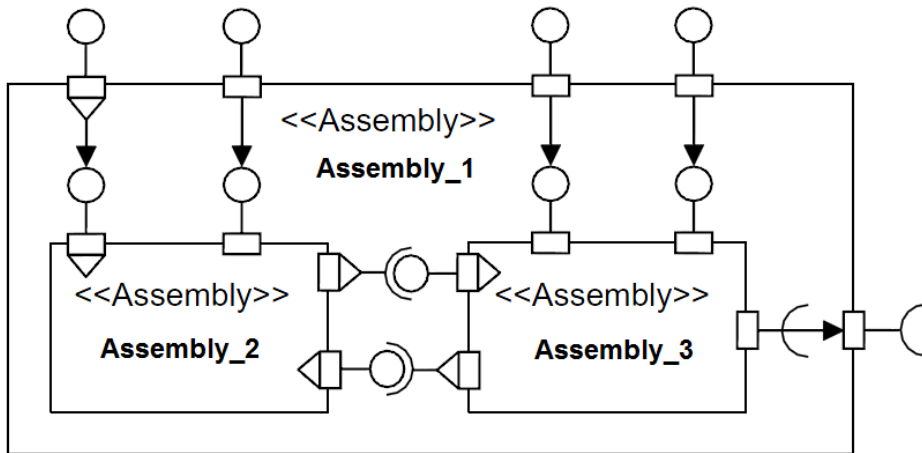


Figure 3.9: A SaveCCM system example.

Chapters 4 and Section 5.1.

SCA offers an environment which takes care of the communication issues (wiring) between components. It does not deal with the implementation details of the components that can be implemented in several object-oriented programming language such as Java or C++. The result is a service-oriented application characterized by a high level of cohesion and by a low level of coupling between components.

Key features of SCA are *components* that can provide interfaces (services), representing the *contract* of the provided services, require interfaces through the definition of *references*, and expose properties. Services and references are connected by links called *wires* in the SCA language. In figure 3.10 an example of a SCA composite with two components inside is shown.

The entire model has an embedded hierarchical nature as each component can be implemented as a software entity developed in a supported programming language or as the result of the composition of a number of sub-components realizing a *composite*.

Components can be composed and configured by means of a simple XML-based language named *Service Component Definition Language* (SCDL). A set of composed and configured components form a SCA application named *contribution*.

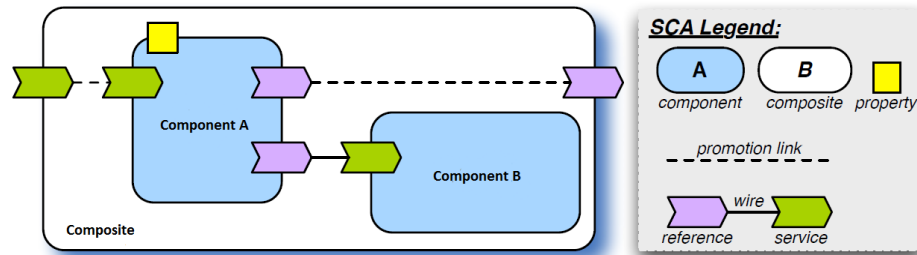


Figure 3.10: SCA components and composites examples.

A SCA composite represent an assembly of components from a logical point of view. The components of a composite can be deployed as an unique process on a local machine or distributed across several process running on a number of networked machines. Moreover, components composing a composite can be implemented by means of several different technologies and programming language and can communicate using different communication mechanisms.

The *binding* concept encapsulates the communication mechanisms across two or more components. When defining references and services, no specification regarding *how* the communication is carried out is given. This is very important since it keeps the component specification completely independent from the adopted communication protocol making their deployment and their reuse easier.

Furthermore, the binding concept allows one of the key-features of the Service Component Architecture: the possibility to interface a SCA application to non-SCA software. A SCA application can interact with non-SCA software, for example a JSP page, a web service client or a plain Java class through remote method call methods such as Java RMI. A SCA application can also access persistent data in several ways using Service Data Objects (SDO) in combination with some standard technology as JDBC or the Java Persistence API (JPA).

As mentioned above, the SCA technology, and the SCA component model itself, do not make assumptions about the underlying implementation tech-

nology that ensures the inter-component communication and the correct deployment of the application. Several SCA run-time environments are available both in free and in commercial form, for example: Apache Tuscany [1] [60], OW2 FraSCAti [9] [72] [73], Fabric³ [2] and IBM Websphere [3].

3.14 Smartsoft

SmartSoft [74] [70] [69] is a component approach for robotic software developed by Ulm University. The SmartSoft component model is based on the intensive use of communication patterns and it is characterized by the explicit support for dynamic wiring of components at run-time. According to the SmartSoft developers, the market of software component for robotics is hampered by the lack of a specific component model targeting robotics, the great challenge of SmartSoft is to provide a generic software architecture without enforcing a particular robot architecture. The SmartSoft component model can be considered as an ADL-like model due to the explicit concept of required interface (*service requestor*) and it can be implemented in several object-oriented programming language. The Corba based reference implementation called *Corba SmartSoft* is written in C++ and uses the *TAO ORB* implementation of Corba. The *ACE* package provides the operating system abstractions ensuring the interoperability across most operating systems.

A SmartSoft component is implemented as a set of threads and interacts with other components via predefined communication patterns. The use of communication patterns let the component to overcome process and computer boundaries. Components can be dynamically wired at run-time using a specific communication pattern.

SmartSoft components interact with each others via predefined communication patterns that define the semantics of the interfaces. A communication patterns also provides predefined access methods, hides the synchronization issues and defines the communication mode that can be one-way, two-way, synchronous or asynchronous. A communication pattern always consists in two complementary parts: the *service requestor*, that can be intended as a required interface, and the *service provider* similar to a provided interfaces.

Ports are interface elements for sending and receiving data encapsulated into *communication objects* which are ordinary objects decorated with additional member functions. They represent the content to be transmitted by communicating components through a communication pattern. This allows to wrap all middleware-related features into communication objects without polluting components with middleware details such as Corba AMI and valuetypes. Communication objects can also be extended using inheritance without affecting the components.

All SmartSoft interfaces are service based, a *service* is a concrete instantiation of a communication pattern and it is represented by the communication pattern used to establish the connection and the communication objects exchanged between components.

SmartSoft provides a set of communication patterns supporting several interaction modes like one-way communication (*send pattern*), two-way communication (*query pattern*), 1-to-n data distribution (*push newest* and *push timed* pattern) and event based asynchronous communication (*event pattern*). The use of communication patterns to manage the exchange of data among components no longer leaves to the component builder to decide whether to invoke a remote service synchronously or asynchronously but defines a freely usable fixed and well tested set of access modes. In particular, the concurrency and the synchrony are fully handled inside the communication pattern without the need of dealing with these issues in each user defined component. The *state pattern* provides a mechanism for coordination among components while the *wiring pattern* supports the dynamic wiring of components.

Only compatible components can be connected, in particular, a service requestor and a service provider are compatible if both the communication pattern and the communication objects types match.

In SmartSoft there is no explicit support for component hierarchy, the architectural configuration of components is defined in the deployment phase when components are connected through communication patterns. Components can also be connected at run-time using the wiring pattern, this is one of the major difference between SmartSoft and other approaches. In order to support the dynamic wiring, a broker service is implemented as a

standalone component. Each service provider can register its services to the broker, after that, the broker returns tuples describing all compatible service providers. These tuples can be used directly by the service requestors to choose a compatible service provider.

3.15 SOFA 2.0

SOFA 2.0 (Software Appliance) [41] [75] is an architecture-based component model developed at Charles University in Prague. This component model is based on its predecessor SOFA but it incorporates a number of enhancements and improvements. In particular it now supports dynamic reconfiguration of components, multiple communications styles and control interfaces managing non functional features like life cycle of components and reconfiguration mechanism.

In the old SOFA component model, the key features were defined by an ADL language. In SOFA 2.0 the ADL is associated with a meta-model based definition. The implementation language proposed is Java, however, the meta-model and SOFA 2.0 abstractions are programming language independent. Each SOFA 2.0 application is executed in a run-time environment called *SOFAnode* which consists of a number of deployment docks providing run-time functionality for executing components. *SOFAnode* provides also a repository containing components descriptions and implementations.

In SOFA 2.0 a component is described by its *frame* and *architecture*. The frame is a black-box view of the component describing its provided and required interfaces. The frame of a component is implemented by an architecture, which represent a gray-box view of the component. A single frame can be implemented by several architectures, at the same time, a single architecture can implement multiple frames. If the component is a composite, the architecture specifies the sub-components and their interconnection on the first level of nesting.

Run-time structure of a SOFA 2.0 component is composed of a *control part*, which consist of the component manager that administrates the component life-cycle, and a *functional part*, which, in case of a primitive component,

contains the code of the component and, in case of a composite component, contains other sub-components implementations.

SOFA 2.0 supports both provided and required interfaces. The type of an interfaces is defined by a signature which can be a reference to an interface definition in the underlying language. All SOFA interfaces are operation-based and they are defined by a set of functions and parameters indicating the services provided or required by the component.

Interfaces can have a *cardinality* parameter indicating the number of bindings that the interface can join. In SOFA 2.0 in fact, both provided and required interfaces can be bound to a number of other interfaces of the same type. Moreover, SOFA 2.0 allows provided-to-provided and required-to-required bindings. This feature is used to “forward” a component interface to sub-components, in particular there is a *delegation* relationship when a provided interface of a component is bound to a provided interface of a sub-component and a *subsubption* relationship when a sub-component required interface is bound to a required interface of the containing component. This characteristic come from the strong focus put by SOFA 2.0 on component hierarchy.

Components are interconnected with bindings among interfaces and bindings are provided with connectors which are first class entities like components. The functionality of each connector is a *communication style*. In SOFA 2.0 four communication styles are supported: method invocation, message passing, streaming and distributed memory. Supporting different communication styles can be an advantage taking into account the run-time environment. From the knowledge of the communication style, in fact, the inter-component communication can be optimized by choosing an appropriate middleware, for example CORBA for remote method invocation or TCP/IP for streaming.

Connectors can be further divided into two categories: *design connectors* and *run-time connectors*. Design connectors are used in the SOFA 2.0 component model to explain the connections between interfaces defining the communication style and other features associated with the interfaces involved in the communication. Run-time connectors are code artifacts used at run-time to implement the above mentioned design connectors. SOFA 2.0

strongly focuses on nested hierarchical architectures.

Each component can be a composition of a number of sub-components and each one of these sub-components can be divided into other sub-components and so on. SOFA 2.0 aims to enhance the support for nested hierarchical structure of components by putting emphasis on dynamic reconfiguration, multiple communication styles and life-cycle management of components.

Component Models Analysis

In this chapter we adopt the analysis and classification model presented in Chapter 2 for the classification of five component models. The analysis is conducted by reviewing the component models along the four main concepts: *Computation, Configuration, Communication* and *Coordination*.

We have chosen five component models: ROS, OROCOS, SCA, CCM and OpenCOM since they can be considered examples of five main approaches in proposing, designing and developing a component model in terms of provided mechanisms and different application domains.

ROS and OROCOS mainly refer to robotics and embedded systems domains. Although their target domains are similar and in most cases overlapping, they present quite different approaches. OROCOS is targeted to hard real-time performances and the main concerns are related to the strict periodic execution of periodic components. ROS, instead, is less focused on real-time performances and, on the contrary, it offers tools and instruments to speed up the development of complex robotic control systems by making available a large collection of reusable implementations and offering to the users a framework of tools for making the design, the development and the debugging of software components easier.

The SCA component model is far more generic and it is much influenced by the Service Oriented Computing principles. The Corba Component Model embodies the Corba inter-process communication principles and mechanisms in a component model. The OpenCOM component model is general-purpose

model although it is well suited for resource-poor environments such as embedded systems. It offers a lightweight run-time kernel and a rich component model that “mediates” the access to the kernel API.

These five component models embody five different approaches. The objective of the classification that we are presenting are twofold: first, we would validate the proposed approach by successfully applying it to five different models, second, we would highlight the similarities and differences between five component models that can be considered iconic of five approaches.

The expected outcome is a classification that can be considered as model for the user who wants to easily analyze and compare several component models in order to identify one or more models that are well suited for his needs.

For each main concepts, we refer to them as the “4Cs”, we explore the corresponding feature diagram highlighting the characteristics of component models by means of present (selected) or non present (unselected) features in the diagram. For each model and for each selected feature we present the constructs, the mechanisms or the tools, both theoretical and practical, provided by the model to address that particular concept. We will summarize the collected results into tables to make the comparison between components easier.

4.1 Communication

Along the communication analysis axis, components can be classified according to three main features: *Synchrony*, *Anonymity* and *Interface* (refer to Section 2.3.2).

4.1.1 Interface

In all component models the *interface* construct is the mean by which the communication between components is carried out. Even though the required interface concept is, in general, not mandatory for a component model, all the five analyzed models allow to define required interfaces. This explicit

definition greatly improves the clearness of the component model in showing the dependencies of each component in terms of needed input data or services.

Component models differ both in interface definition and semantics. First of all, even if the underlying concept is almost the same, different models use different names to define both provided and required interfaces: provided interfaces are called “facets” in the Corba Component Model (CCM), “services” in SCA and simply “interfaces” in the OpenCOM component model. This difference is reflected also in the required interface concept since they are called “receptacles” in CCM and in OpenCOM and “references” in SCA.

ROS and OROCOS slightly differ from other component models since their interface concept is embodied by topics and ports respectively. Subscribed topics can be considered as required interfaces for ROS while published topics are provided interfaces. In OROCOS the input ports and the event input ports can be classified as required interfaces while the output ports are provided interfaces. These definitions can be explained by interpreting the provided interfaces as constructs by which a component make its computation services available to other components. Similarly, required interfaces represent dependencies of the component to carry out its computation or task. Hence, from this point of view, input ports and subscribed topics represent sources of data needed by the components while output ports and published topics represent data output used by the component to deliver the results of its computation.

It must be noted that, along with the ports and topics constructs, both ROS and OROCOS provide the concepts of service and operations respectively, we will discuss about it hereafter.

Component models interfaces can be constructed in different ways. Regarding this feature we can identify the deepest differences in component models interface semantics. The operation based construction represents the operation-oriented semantics in communication and data delivery between components. For CCM, OpenCOM and SCA this is the only allowed mechanism for defining interfaces. In SCA, above all, this concept is deeply routed into the followed service-oriented approach. When this semantics is adopted, components offer their functionality to other components, and

require functionality from others, in terms of callable operations. Models may however differ in terms of provided communication mechanisms and in decoupling level but the underlying semantics involved in operation-based interface concept does not change.

In ROS and OROCOS the communication semantics, and hence the interface construction, is deeply different. The focus is shifted to the pure data exchange mechanisms: components behaves as data producers and consumers rather than operation callers and providers. From this point of view, topics and ports represent two different artifacts for implementing the same underlying approach. The provided services of a component are delivered in the form of computation results provided by a component to another, on the other hand, the dependencies of a component from others are expressed in terms of needed input data rather than needed services.

However, the data-oriented semantics embodied by topic-based and port-based interface construction can make the request response interaction scheme quite tricky to implement as some forms of request/response mechanisms should be provided *above* the port or topic concepts. In this case there is the concrete risk of loosing the intrinsic simplicity and flexibility of the data-oriented approach. For this reason, both OROCOS and ROS offer means to carry out some forms of request/response-oriented communication. In OROCOS this is embodied by the concept of *operations* provided by a component and callable by others. Operations are plain C/C++ functions running in the provider thread or in the caller thread and can be grouped into service objects. ROS services are implemented as message couples (one for the request and one for the response). A component can advertise a service making it available to other components that can invoke it by sending the appropriate message and receive the response through another message. Each service has an interface that defines the function signature that must be known to components that want to call it. It must be said that both the ROS services and the OROCOS operations should be considered as extensions to the standard communication mechanisms represented by topics and ports respectively.

Interfaces can be defined by using the same programming language adopted

for the component implementation or by means of a specialized Interface Definition Language. The use of an IDL makes the interface definition independent from the adopted programming language and helps in keeping the component implementation separated from its interface definition. However, the use of an IDL can differ from a component model to another.

CCM and OpenCOM refer to OMG-IDL for the definition of component implementations, so doing, the interface definition is totally independent from the chosen implementation language.

ROS defines topic names in the component implementation code although these names can be remapped when deploying the components as described in Section 4.3, on the other hand, messages are defined by means of the *Message Definition Language* (MDL) that is a simplified strongly typed IDL adopted for defining message structure and fields data types. Similarly, services are defined by means of the *Service Definition Language* that is an extension of the MDL defining the couple of messages involved in the request/response interaction.

In SCA interfaces can be defined in several ways. The composite file, written in the SCDL XML-based language, defines the names of references and services of each component appearing in the composite. Each one of these interfaces often refers to an implementation language file containing the details of the implementation. If the implementation language is capable of defining interfaces, like Java, this can be used for the interface definition, if not, a specific interface definition language can be used instead. For web-service interfaces, for example, the *Web Service Definition Language* (WSDL) is typically exploited.

OROCOS is the only one of the five analyzed models that mainly defines the interfaces (ports and operations) in the component implementation code. Being the OROCOS component implemented in C/C++ language, the ports and operations definitions are done in the source header files. In the deployment file, however, ports and operations are also mentioned and, in the same file, they are wired with other components.

4.1.2 Anonymity

The anonymity feature is related on the amount of information that one component must know for interacting with other components. The anonymity concept is deeply routed into component-based software engineering since it is one of the key features that makes single components independent from others and reusable. For this reason most component models provide means for making the communication between components anonymous, however, the level of anonymity can be much variable from a component model to another one.

In general, data-oriented mechanisms should provide a greater degree of anonymity since, during the interaction, the focus is on the data exchanged, no matters which component is producing or consuming it. For operation-based mechanisms a certain level of knowledge is needed for invoking the right service, in these cases, this knowledge is held by the run-time system associated with the component model. Interacting components may only need to know each other interfaces to interact ignoring the details of the components instances.

The Corba Component Model allows two kinds of interactions with different levels of anonymity. When event-based interaction is adopted, components can generate events (event sources) and/or react to events (event sinks). Events are transported by an *event channel* provided by the Corba framework that lets the components to publish and consume events. Event consumers components only need to know which event channel is transporting events and subscribe to it. This mechanism provide a fully anonymous communication although this possibility is restricted to events. When communication is carried out through operation-based interfaces (facets and receptacles), the operation caller, or client, needs to have some information about the component instance that it is providing the invoked operation (server). In CCM this information is hidden by the underlying Corba framework that provides to the client the *stubs* objects. Stubs are automatically created by the Corba infrastructure by interpreting the IDL-based definitions of components interfaces, this way, the information needed by a client to exploit functionality offered by a server

is restricted to the server interface.

Similarly, in OpenCOM interfaces are defined by means of OMG-IDL files. The binding between a provided interface (*interface*) and a required interface (*receptacle*) is provided by binding components that are OpenCOM components specialized in providing the connections between cooperating components. The binding components can be manually created by the developers or dynamically loaded by *binders* from a set of already defined bindings. Binders exploits low-level binding features provided by the OpenCOM *kernel* for connecting interfaces to receptacles. The kernel is in charge of interpreting the IDL definitions giving to the binding components the correct instances of the communicating objects. This mechanism allows OpenCOM components to obtain a level of anonymity that is comparable to the anonymity offered by Corba since components only need to know each other interface definitions while the binding components keep the interfaces decoupled.

In SCA the anonymity is guaranteed by the run-time environment that interprets the interface definitions provided by composite files referring, if necessary, to separate interface definitions. When defining a reference, components only need to know the name of the service, as it is defined in the composite file, and the interface definition since the current component instances are fully managed by the run-time.

In ROS, when topic based communication is exploited, components, called *nodes*, only need to know the URI of the topics and the associated message definition. URIs are expressed as plain strings that must be unique in the entire ROS system. When a node interact with a topic by publishing or subscribing, it totally ignores the presence of other nodes interacting with the same topic. The communication is managed by the ROS master node that exploits the transport mechanism offered by the ROS run-time system. The transport mechanism is based on TCP/IP sockets (TCPROS) or UDP/IP communication (UDPROS).

When service-based communication is adopted in ROS, communicating components only need to share information regarding the interface of the service operation that is provided or required and the format of the request/response messages. In a C++ implementation, for example, the interface of the

operation is stored in a C++ header file defining the prototype of the function while the messages formats are defined in special files that are automatically translated into executable code by the ROS framework.

In OROCOS the port-based communication style allows total anonymity between components. The information needed to connect input to output ports is stored in the deployment file or script used by the OROCOS *deployer* to actually connect components. For this reason, the write operation on an OROCOS port is said to be “send-and-forget”. Anyway, the anonymity level reached with the operation mechanism is lower since the invoking component must explicitly obtain the exact reference of the operation provider instance by the run-time system.

The concepts related to the configuration files used by the component models to build a system configuration will be thoroughly analyzed when discussing the configuration concept in Section 4.3.

4.1.3 Synchrony

The synchrony feature is related to the time coupling between the interacting components. When the communication is carried out by means of ports or topics, the communication is always asynchronous. This means that the data producer component do not need to stay blocked while the consumer reads the received data. On the other hand, in the analyzed component models, operation-based interfaces usually provide synchronous interaction, with some exceptions.

In CCM, facets allow components to expose interfaces that can be invoked synchronously via Corba’s two-way operations or asynchronously via Corba’s asynchronous method invocation (AMI) that are part of the Corba 3 framework. The desired mechanism is selected by the developer according to the nature of the particular provided operation. The event-based communication provided by CCM is, instead, totally asynchronous.

In OpenCOM no particular means for handling the synchrony of invocations is provided. Moreover, since the binding components usually mediate the operations invocations, both mechanisms can be provided for the same

interface by simply changing or reconfiguring the binding component.

In SCA the interaction is synchronous by default although an asynchronous behavior can be obtained exploiting the *callback* mechanism. A callback interface is composed of a bidirectional interface, one part is for the request and the other side is for the callback. If using callbacks, a client component can request a service through a reference and continue its flow of execution. At the same time, the server component processes the request and, when its computation is finished, it sends back to the client the response via the callback interface.

In ROS, the topic communication is strictly asynchronous since a component can send a message on a topic and continue its execution. The subscribed components will then be notified by the ROS system and fetch the message from the subscribed topic. On the other hand, when synchronous communication is needed, ROS provides the service mechanism that it is strictly synchronous. Once a component sends a request for a service, it will be blocked until the service provider will send back the response.

The port-based communication of OROCOS components is asynchronous since components can write on output ports and continue the execution. Data on input port is retrieved by polling the port periodically. If event input ports are exploited, an event can be generated when new data arrive on the input port triggering the component's `updateHook` method or function. In analogy with ROS services, OROCOS operations allow synchronous interaction since the operation requester can wait for the completion of the called operation handled by the operation provider.

4.2 Computation

Computation is related to the components nature, definition and implementation. All the five analyzed component models support one or more concrete implementations since we decided to analyze “real-world” component models and not models that are only proof of concepts or theoretical proposals.

The computation concept can be roughly divided into four main features: the *component implementation language*, the *component behaviour*, the *real-*

time support and the *distributed system support*.

4.2.1 Component Implementation Language

For what regards the component implementation languages adopted, analyzed component models can use an ADL-like language [63] or directly use one of the supported programming language. When using the programming language for the definition of components, we can identify two cases: only one programming language is supported or multiple programming languages are available. Although the use of an ADL-like language obviously promotes the support for multiple implementing language, we can observe that component models that defines components by means of the implementation programming language can still support several different languages.

In OROCOS, components are defined by means of one the two supported programming languages which are C and C++. Also in ROS components are defined by means of programming languages, however, the ROS approach is slightly different: instead of providing a multi-language run-time environment, ROS proposes several versions of the same environment for natively supporting different languages. The same *client library*, that provides the low-level instruments to create and manage nodes and topics, is distributed in all the supported programming languages that are C++, Java, Python, Lua and LISP. Once separately compiled, components developed in different languages can seamless interact with each other and with the run-time environment.

The SCA component models theoretically supports all possible object-oriented languages for component definition. The actual support is however limited by the adopted run-time environment. Apache Tuscany supports Java, C++ BPEL and Spring languages and it is distributed in two versions: one is implemented in Java and the other one is implemented in pure C++ and it is often called *native*.

The OpenCOM component model defines *component types* by means of an extension of OMG-IDL. Once a component type has been defined and associated to a component implementation, it can be instantiated becoming a *component*. OpenCOM provides extensions for other definition languages

such as ACME-like and XML based languages, moreover, the extension APIs can be exploited to provide support for other definition languages. Several native kernel implementations are available for a number of programming languages such as Java and C++.

The Corba Component Model has an hybrid approach: components can be defined by means of a Corba-supported object-oriented language or can be defined with the *Component Implementation and Definition Language* (CIDL). These definitions are then translated by the CIDL compiler into the so called *executors*. Executors contain auto-generated implementation stubs and provide hook methods that can be filled by developers to define specific components behaviours.

4.2.2 Component Behaviour

The component behaviour feature refers to the presence of a default Finite State Machine (FSM) that regulates the behaviour of components. Among the five analyzed component models, the only one that has a mandatory FSM definition is OROCOS. Each OROCOS component must implement a default three-state finite state machine and the behaviour specification of each component is actually carried out by specifying the operations that have to be executed during the state transitions of the machine as presented in Section 3.6. This forced behaviour specification reflects the fact that OROCOS was designed having in mind the embedded and control systems domains in which common components behaviours, for example a controller, can be easily defined by means of finite state machines.

In SCA no default state machine need to be implemented but component initialization and destroy operation can be specified. This can be useful to allocate and free particular data structures needed by the component or to save and retrieve persistent data.

OpenCOM, ROS and CCM give to the developers the freedom to fully define the behaviour of components without having to be compliant with any default state machine specification.

4.2.3 Real-time Support

The support offered by the component model can be useful when having to cope with systems that are constrained on the response time. We defined the differences between *soft real-time* and *hard real-time* concepts in Section 2.3.3.

Among analyzed component models, the only one that offers an explicit support for hard real-time systems is OROCOS. The component specification includes the definition of an execution period and a priority level for each component and the run-time framework schedules the execution of the set of instantiated components in order to execute components at the correct frequencies meeting all the deadlines. The OROCOS framework can be compiled both on standard and real-time operating systems, it is obvious that hard real-time performances can be achieved only when exploiting the real-time support offered by an underlying real-time operating system. OROCOS is compatible with several real-time operating system such as RTAI patched Linux distribution [11] or Xenomai-based systems [25]. However, when compiled and installed on standard operating systems, the OROCOS framework can deliver good performances in terms of delay time, periodic execution and repeatable behaviour. The OROCOS framework installed on a non real-time system is suitable when soft real-time performances are needed.

The ROS component model allows the definition of the execution frequency of components, however, no specific support for hard real-time performances is provided. However, the small memory and computation footprint of the run-time system allows for good overall performances, hence, ROS can be effectively adopted when there is no need for high execution frequencies in soft real-time systems.

On the other hand, neither SCA, nor CCM, nor OpenCOM offer explicit support for real-time execution.

4.2.4 Distributed System Support

All the analyzed component models provide means to distribute the execution of components across several networked computation nodes.

In SCA this is achieved by means of several kinds of available bindings between references and services that allow to connect components residing on different machines. One of the possible bindings, when using Java as implementation language, is the Java Remote Method Invocation (RMI). The use of *web services* encourages the distribution of services across the network, these services, described in WSDL, can then be individuated by means of the UDDI protocol and invoked by using the SOAP protocol.

In OpenCOM, components can be transparently distributed on a network since the communication can be managed by binding components provided by binders elements. A binding component mediates the communication between two component in such a way that it is invisible to the components lying at the opposite ends of the communication.

The Corba Component Model provides a standard technique that developers can apply to make the packaging and the deployment of components across multiple servers easier and clearer. CCM components can be described by means of the *Open Software Description* (OSD) language which is a XML Document Type Definition (DTD) defined by the W³C. Components are then packaged into assembly files and OSD files are used to describe the content of each assembly file and its dependencies.

OROCOS allows the execution of components across distributed nodes by exploiting the supported Corba framework. The hard real-time performances, in this case, can be penalized due to the inevitable delays introduced by the networked communication and by the Corba infrastructure itself.

ROS natively supports the definition of distributed systems since the underlying transport service, that manages both the topic-based and the service-based interaction, is based on TCP/IP or UDP/IP protocols. By doing so, the communication between local or remote nodes can be handled without distinction given that components reside in a common network address space. Only one *ros core* application must be running at the same time on a chosen master node of the ROS system; all other nodes need to know the network address of the master node.

4.3 Configuration

Configuration defines the architecture of a system in terms of which components are present and how they are interconnected (refer to Section 2.3.4). Along this axis, components can be classified according to four main features: *configurable entities*, the adopted *configuration model*, the *composition mechanism* and the *persistence mechanism*.

4.3.1 Configurable Entities

Along this feature the five analyzed component models are almost similar. In all component models both properties and connections among components can be configured, however, some remarks can be done regarding the component states configuration. We will discuss about the mechanisms provided by the models for the static and dynamic configuration in Section 4.3.2.

The possibility of configuring the connections among components embodies one of the basic principles of component-based software engineering, in fact, starting from a predefined set of components, different systems can be realized by simply interconnecting components. In the Corba Component Model these interconnections are obtained by binding provided interfaces and required interfaces called *facets* and *receptacles* respectively. Other connections may involve the link between event sources and sinks by means of *event channels*.

In OROCOS, input ports are connected to output ports building a graph of interacting components, the same can be said for ROS in which components can interact by means of subscribed and published topics matched in function of their URI.

In SCA, inter-component connections are handled by coupling services with references, these links may, or may not, cross the composite boundaries in a way that it is transparent to the component implementation. In OpenCOM, a binding component is usually adopted to mediate the communication between components but, from a configuration point of view, a component-to-binding connection is managed exactly as a component-to-component connection since binding components are fully fledged components.

In all analyzed component models, components can expose parts of their

implementations, typically fields, parameters or variables, so that their value can be set from outside. We refer to these as properties (*attributes* in the Corba Component Model). The possibility of setting component parameters greatly improves the flexibility and the reusability of component implementations, we will discuss the mechanisms of setting the properties in Section 4.3.2.

The possibility of setting up and configuring the component state is strongly related to the component implementation defined by the model. Neither OpenCOM nor ROS nor CCM allow the configuration of the component state simply because no predefined state machine needs to be implemented by default within the component implementation. Of course, it is possible, for the developer, to define a FSM that manages the component behaviour and the initial execution state of this machine could be set by a property value, however, this semantics is external to the component model and hence it is not considered in our classification.

Due to the strong state machine oriented semantics of OROCOS components, the initial state of each component's state machine can be externally set in the configuration phase. In particular components can be started by triggering their start transition or stopped by triggering the stop transition or can be put into a preoperational state in which they can be reconfigured. These state transitions are managed by the deployer agent.

The SCA component model offers, to some extent, the possibility of configuring the state of a component by exploiting the *scope* concept. A component can be configured to have a *stateless scope*, a *composite scope* or a *conversation scope*. For Stateless-scoped components a new instance of the component is created on each service call, for composite-scoped components, a new instance is created on the first service call and then it is used to serve all the subsequent service calls, for conversation-scoped components, a new instance is created for each conversation that represents a series of correlated server-client interactions. Besides this, in the component implementation, particular methods can be marked to be called on instance creation and on instance destruction allowing the developer to manage state and persistence features of the component.

4.3.2 Configuration Model

The configuration model feature refers to the mechanisms provided by the component model to support the configuration of the system. On the basis of which kind of mechanisms are provided, component models can support the static or dynamic configuration, and reconfiguration, of components. When dealing with the static configuration, we identify two main phases: the compile-time and the deploy-time. The five analyzed component models strongly differ with each other when compared along the configuration model feature.

In the Corba Component Model the configuration can be carried out both at the deployment phase and during run-time. The component configuration is managed by a *component server* which is in charge of interpreting Interface Definitions Language (IDL) files and Component Implementation Definition Language (CIDL) files in order to set up the components by creating stubs and skeletons and their connections. This operations can be done at deployment time. For the run-time configuration, CCM offers a number of possibilities to the developers. Each component expose a *home* interface that can be used by other components to manage the component life-cycle, for example to create and remove components instances. The references to all the home interfaces of the available components are stored in a centralized database that can be accessed by clients. Components can be deployed in component servers that use the *standard configurator* interfaces to configure components. Component developers can extend this configuration interface to specify the custom configuration operation allowed for the particular component. The *port* construct is used to enable configuration mechanisms by creating component connections, establishing component attributes and subscribe or publish events.

In the OpenCOM component model the configuration of components is managed by the kernel APIs. These kernel interfaces can be directly called by component implementation for configuring itself or other components. However, especially when run-time configuration is needed, OpenCOM provides a rich infrastructure to manage the components connections, their properties

and the overall system structure. Components can be collected into *component frameworks* (CF) that are set of correlated components. A set of components, or component frameworks, can be executed within a *capsule* and, optionally, components can be included into *caplets* constructs. The caplet concept itself is made available by means of a particular component framework called *caplet CF*. Within a caplet, components can be dynamically loaded, unloaded and instantiated exploiting the caplets' API that avoid to directly access the underlying kernel APIs for configuring components. Capsules, Caplets and Component Framework are parts of the very rich OpenCOM component model thoroughly analyzed in Section 4.2.

OpenCOM provides the concept of *reflective extensions* to support the construction of *dynamic* target systems that need to change or evolve during their execution in a controlled and principled manner [45]. This is achieved by means of three extensions called *interface meta-model*, *architecture meta-model* and *interception meta-model*. The interface meta-model provides the capability to discover at run-time the details of the interactions between components, expressed as operation signatures, and to invoke these operations. These capabilities enable the components to invoke operations that are not known at deployment time, concretely altering the system configuration. The architecture meta-model represents the topology of the actual set of components and can be used to inspect the topology and the structure of the system with the possibility to change inter-component connections and add new components. The interception meta-model give access to the process of invoking an operation in a component interface. This allows the developers to add a special object, called *interceptor*, that intercept operation invocations between components triggering side-operations such as data logging or routing the request to other components. When this last possibility is exploited, this can concretely change the component interaction structure at run-time by adding other interacting components.

In SCA the system configuration is carried out at deployment time. Configurations are defined in terms of allowed components, reference-to-service wires and component properties. All these information is stored in the XML-SCDL composite files that represent the main instrument offered by SCA to configure

a system. Wirings between services and references can be user-defined or system-defined. In the first case the connection is defined by developers who manually match services and references, in the second case the SCA run-time is in charge of connecting the interfaces according to their definitions and types.

The SCA standard defines mechanisms and guide-lines to allow the run-time reconfigurability of a system but the actual possibility of dynamic configure and re-configure systems depends on the run-time implementation that it is adopted. Apache Tuscany does not provide mechanisms for explicitly reconfigure a system that has been instantiated while, on the other hand, OW2 FraSCAti run-time allows for reconfigurability and component life-cycle management as presented in Section 5.1.11.

In OROCOS, components interfaces are defined above all at compile-time when ports are instantiated. The configuration of port bindings is done at deploy-time by the OROCOS *deployer* which is in charge of running components by loading an XML-based deployment file or by executing a deployment script. Component properties are stored in XML files associated to components by means of a particular element in the deployment file. The possibility of re-configuring a system during run-time is rather limited: component states can be changed by the deployer at run-time in response of malfunctioning, by following scripted instructions or by directly interacting with the user through a command line interface. When components return in the pre-operational state (after being stopped) they can be re-configured by the deployer.

In ROS, the publisher-to-subscriber and the services bindings are carried out on the bases of their respective URIs. The name of a topic or of a service is defined in the component implementation code, hence, the configuration is mostly defined at compile-time. However, at least for topic names, a remap service is available and allows to change the topic names at the system startup. This is achieved by means of XML *launch files* containing information about the graph on nodes which are present in a system, the topic names and remappings, and the properties values. Once the system has been started, no further configuration operations are possible except the fact that ROS nodes,

being autonomous processes, can be started, killed and restarted by users at run-time allowing a minimal possibility of dynamic system reconfiguration.

4.3.3 Composition Mechanism

Not all the analyzed component models allow the composition of components inside structures. In ROS, the stack and package organization is only at the host filesystem level, once executed, components are organized in a flat graph of nodes without any containment concern. The same can be stated for OROCOS in which component cannot be collected in higher-order structures.

In the Corba Component Model, components can be wrapped inside a *container* element. Containers contain exactly one component and their role is twofold. First they expose component interfaces (facets, receptacles and the home interface) without adding any computation (facade), second, they mediate the access to both configuration related interfaces, such as ports, and to callback interfaces used to capture events. In this second case the container shows an adapter pattern behaviour. However, containers are only wrappers around single components and they do not provide mechanisms for organizing components into higher-order structures.

For this reasons, the only component models that present a complete composition mechanism are SCA and OpenCOM.

In SCA components are collected into composites that do not add any kind of computation to the component set. Their role is to expose a subset of the components interfaces to be accessed from the outside. We classify this approach as a facade pattern.

In OpenCOM the composition mechanism is far more complex. Components can be collected into component frameworks, according to Coulson et al.: “a component framework is a tightly-coupled set of components that *i*) cooperates to address some focused area of concerns, *ii*) provides a well-defined extension protocol that accepts additional ‘plug-in’ components that modify component framework’s behaviour *iii*) constraints how these plug-ins may be organized” [45]. In this sense component frameworks expose to the outside a subset of the inner components’ interfaces and, at the same time,

can mediate the access to other parts of the interfaces. The former approach can be traced back to a facade pattern approach, the second is most similar to the adapter pattern.

OpenCOM provides the concept of *capsule* as containing entities in which components are loaded, instantiated and composed, the role of a capsule is to give access to OpenCOM kernel API. Components contained in capsules, however, are not subjected to any hierarchical composition. Finally, in OpenCOM, components can be associated to caplets that can offer an insulated environment to components within a capsule. Each capsule can contain a primary caplet that can contain several other caplets called *plug-in extensions*, each one of them is in charge of managing one or more components. Being the OpenCOM concepts implemented with the OpenCOM component model itself, caplets are made available within the *Caplet Component Framework*.

4.3.4 Persistence Mechanism

The persistence feature is related to the possibility of saving and retrieving a particular configuration or, if possible, the state of the system. For that regards the persistence of component properties and connections, all component models provide the same mechanisms that are used for the configuration of a system, these are: XML-SCDL composite files in SCA, launch files for ROS, IDL and CIDL files for Corba Component Model, IDL and related extensions for OpenCOM and deployment files or scripts for OROCOS. The possibility to store and retrieve the *state* of a system is only offered by Corba Component Model through a set of persistence APIs of the *Component Implementation Framework*. These APIs manage the persistent state of components in a system and can be used to construct the implementation of a component-based software system. The *Component Implementation Definition Language* adopted by CCM allows to describe, besides the implementation details of a system, also its persistent state. This feature is useful when CCM is exploited to implement entities that have to persist over time and are usually represented as database entries, such as bank accounts. Components that need for a persistent state are mapped to a persistent data storage system

(e.g. a database) that can be used to reconstruct the state of the system.

4.4 Coordination

The coordination concept is concerned with the interactions among components and their roles and can be divided into two features: the *connector* feature and the *component roles*.

4.4.1 Connector Feature

Connectors are special kind of components specialized in managing the interactions between other components. Connectors are present in all component models, the main difference is that in some models connectors are first-class entities with a special semantics while in other component models their presence is hidden. The five analyzed component models reflect this distinction.

In CCM connectors are not first-class entities since the handling of the interactions among interfaces is provided by the underlying Corba framework by means of the stub-skeleton mechanism. However, developers can provide their own connectors by developing components that can be placed between a server and a client to control and manage the communication.

Also in ROS, connectors do not appear as first-class entities since the communication is fully handled by topics or services and no additional semantics can be added. In particular topic-based communication embodies the publisher/subscriber semantics while the service-based communication offers a client/server semantics.

In OpenCOM, instead, the binding components embody the concept of connector. Bindings are provided by binders specialized components and fully manage the communication between interfaces and receptacles.

The SCA component models clearly separates the service to reference association from the semantics associated to it. The first concept is embodied by the *wires* while the second is provided by the *binding* concept. Bindings specify the modality through which components can communicate.

In OROCOS the connector feature is given by the *connection policy* associated to each port-to-port link.

Since OpenCOM, SCA and OROCOS provide a concrete concept of connector, we can analyze the coordination tasks associated to these constructs. In these component models, and in others too, connectors are used to define coordination policies between components as explained in Section 2.3.5. The identified coordination tasks are: *interface adaption*, *data filtering* and *ordering and protocol* operations.

OpenCOM binding components can be adopted to implement a great variety of functionality: the interface adaption is the most common since the binding can connect itself to an interface and, at the same time, expose to other component a modified version of the same interface, for example varying the data types or the method signatures realizing an adapter pattern. Ordering, access protocols and filtering can be implemented as well although, in the OpenCOM specification, the implementation of lossy protocols is strongly discouraged and connection between interfaces and receptacles are intended to be “reliable” in all cases.

SCA bindings can be exploited to implement ordering protocols for managing multiple invocation of the same services and, in many systems, the bindings are in charge of implementing non-functional features like atomic transactions, security, cryptography and so forth.

OROCOS connection policies defines data buffering, locking mechanisms and initial state of the communication. The connection policy also allows to specify a transport mechanism for data, for example the Corba transport can be needed for ensuring the communication between distributed components.

4.4.2 Component Roles

Another aspect connected with the coordination concept is the presence of means provided by the component model for a clear separation of roles between components in a system. Two kinds of separation of roles can be identified: *client/server* relationship and *master/slave* relationship. In case no means for a separation of roles are provided, components can be considered

as equals entities involved in a *peer-to-peer* interaction.

All analyzed components provide a service-based interaction scheme in which some components provide services through their provided interfaces and other components require services through the required interfaces. This mechanism embodies a client/server relationship in which the service provider plays the role of the server with respect to the components that request the services that are its clients. However, even though this mechanism is provided by all analyzed component models, it is not the principal interaction scheme for some of them.

In ROS and OROCOS, components mainly communicate via topics or ports in a peer-to-peer fashion where one or more components produce data that it is consumed by consumer components. The Corba Component Model provide a publisher/subscriber event-based communication semantics by means of event producers and sinks. This communication pattern do not define any separation of roles between interacting entities and, hence, it can be classified as a peer-to-peer interaction.

4.5 General Considerations

From the analysis conducted so far we can correlate the presence, or the absence, of particular mechanisms and constructs offered by component models with their specific application domain. The five analyzed component models can be ordered on the basis of their application domain: SCA and CCM are fully general purpose models, although the possibility of building generic applications is pursued by means of different approaches. SCA exploits the concepts of Service Oriented Applications (SOA) modeling interactions as services, while, CCM offers a component-based extension to the Corba framework that it is based on remote method invocation and interoperability among object-oriented software entities.

On the opposite side, OROCOS is a fully specialized component model developed keeping in mind the needs of hard real-time embedded systems, such as controllers and data acquisition systems. ROS can still be classified as a domain specific component model since it is strictly addressed to the robotic

control system development. The robotic control domain is still focused on control and data acquisition but, at the same time, it spans from low-level controllers to high-level software for motion planning, mapping, navigation and SLAM. For this reason, ROS is less narrowly targeted on low-level control software and provide means to design and develop more generic applications involving, besides controllers, higher-level applications.

The OpenCOM component model lies in the middle: it is intrinsically a general purpose component model since it does not provide instruments for accomplishing typical controller or hardware related tasks such as the possibility of imposing periodic execution of components. Yet, it is still intended for embedded software domain spanning from data acquisition systems to network devices. For this reason it provides a lightweight kernel implementation and the possibility to deploy OpenCOM systems both on general purpose machines (PCs) and on application specific devices, like network processors and microcontrollers.

This difference in application domains is manifested in the constructs and mechanisms provided by the component models for addressing the four areas of concerns that we identified: the communication, the computation, the configuration and the coordination.

The interface nature of the analyzed component model reflects the chosen communication style of each model. OROCOS ports and ROS topics represent constructs for data-oriented communication that is typical for embedded system oriented component models. Operation based interfaces are represented by means of services, interfaces or facets depending on the particular language adopted by component models. All component models, except for OROCOS, use some kind of interface definition languages for expressing the presence and the general information associated to interfaces, being them ports, topics or operation-based. This makes the definition of the component interfaces independent from the implementation language. The fact that OROCOS directly defines interfaces in the source code of components is related to the fact that it supports only C/C++ implementation languages for components.

Anonymity in inter-component communication is a widely achieved feature in all analyzed component models since it makes components more

	COMMUNICATION									
	Synchrony		Anonymity		Interface					
	Synchronous	Asynchronous	Anonymous	Non-Anonymous	Type		Construction		Definition	
	-	-	-	-	Provided	Required	Operation-based	Data-based	Programming Language	IDL
Corba Component Model	Corba two way operations	Corba AMI	events sinks and sources	facets and receptacles	facets	receptacles	facets and receptacles	events sinks and sources		OMG-IDL
SCA Component Model	services	callbacks	only interface information needed		services	references	services and references		O. O. programming language	SCDL
OpenCOM Component Model	user defined	user defined	bindigs		interfaces	receptacles	services			OMG-IDL
ROS Component Model	services	topic-based communication	topics and services (only interface information needed)		publish on topics, service advertisement	subscribe to topics, service call	Services provided and required	topics		MDL and SDL
OROCOS Component Model	operations	ports	ports	operations	output ports and operations	input ports and input event ports	operations	ports	C++	

Figure 4.1: Communication summary table.

independent from each other and, hence, more reusable.

The asynchronous communication mechanisms is provided, above all, by embedded system specific component models since it helps in developing time constrained software systems. The possibility of achieving synchronous communication, when needed, is still provided both by ROS and OROCOS via services and operations respectively. Figure 4.1 summarizes the communication features.

All component models, except for OROCOS, provide support for multiple implementation languages. General purpose component models support the presence of multiple languages by providing a framework that can natively execute heterogeneous component implementations, on the other hand, OpenCOM and ROS provide several implementations of the same run-time environment in different programming languages. This choice should be put into relation with the goal of achieving good run-time performances together with a small footprint for the run-time environment. These features can be

very important when developing software for resource constrained embedded systems. Along with this feature, the support for real-time execution is provided by components which are focused on embedded system domain. In this case, OpenCOM do not explicitly provides means for real-time execution even if the lightweight kernel implementation could be exploited for obtaining low execution latencies on resource poor systems. General purpose component models exploit the use of ADL-like languages for defining components in a language-independent way improving the independence from any implementation technology.

The OROCOS component model is the only which constraints the implementation of components into a default FSM. This, again, is related to the strong focus of this model to the controllers and embedded system domains in which the greatest part of the control tasks can be effectively modeled by means of finite state machines.

COMPUTATION								
Component Implementation Language				Component Behaviour		Real-Time Support		Distributed System Support
ADL-like Language	Programming Language			Customizable Component State Management	Predefined State Machine	Soft Real-Time	Hard Real-Time	-
-	Multiple Language Support	Single Language Support		-	-	-	-	-
Corba Component Model		Multiple languages		user defined				yes
SCA Component Model		C++, Java, BPEL and others		user defined				yes
OpenCOM Component Model	extended OMG-IDL or user defined languages	many supported		user defined				yes
ROS Component Model		C++, Lisp, java, Python, Lua...		user defined		supported	only by integration with other frameworks	yes
OROCOS Component Model			C/C++		model provided FSM		if R-T O.S. is provided	yes

Figure 4.2: Computation summary table.

The support for distributed systems is provided by all component models since, nowadays, systems are often distributed across several computation nodes. This is true both for general purpose systems, such as Internet-based applications, and for embedded systems where the control tasks are

carried out by a set of networked heterogeneous nodes such as standard PCs, microcontrollers, network processors and so forth. In Figure 4.2 computation features are summarized.

Regarding configuration, all component models, regardless of the specific domain they are intended for, offer file-based mechanisms for defining and making persistent the configuration of a component-based system, however, they still differ in adopted languages and formalisms. General purpose models offer mechanisms for dynamic reconfiguration of a component system while this feature is not so supported in domain specific component models even though a limited possibility for stopping and restarting components is still provided. Figure 4.3 summarizes the configuration features.

	CONFIGURATION											
	Configurable Entities			Configuration Model				Composition Mechanism		Persistence Mechanism		
	Properties	Component States	Connections	Static Configuration		Dynamic Configuration	Facade Pattern	Adapter Pattern	Properties	Component States	Connections	
	-	-	-	Compile-time	Deploy-time	-	-	-	-	-	-	
Corba Component Model	CIDL and XML-OSD files		CIDL and XML-OSD files			configuration interface	containers	containers	CIDL and XML-OSD files	CIDL and XML-OSD files	CIDL and XML-OSD files	
SCA Component Model	composite files		composite files		composite files	allowed (not always implemented)	composites		composite files		composite files	
OpenCOM Component Model	OMG-IDL		OMG-IDL		allowed	reflective meta-models and specialized components	capsule and caplets	capsule and caplets	OMG-IDL		OMG-IDL	
ROS Component Model	launch files		launch files	hard-coded topic names	remapping in launch files				launch files		launch files	
OROCOS Component Model	deployment instructions files	deployment instructions files	deployment instructions files		deployer configuration	properties and states			deployment instructions files	deployment instructions files	deployment instructions files	

Figure 4.3: Configuration summary table.

For what regards composition mechanisms the distinction between general purpose and domain specific models is blurred. ROS and OROCOS do not provide means to organize and compose components into hierarchical structures, general purpose models, instead, offer composition mechanisms but each one of them present strong differences in the adopted approach. CCM containers can contain only a single component and they are intended for facilitating the life-cycle management of the component, SCA composites

are containers that collect related components by simply hiding or exposing parts of their interfaces, without adding any kind of computation or additional services. OpenCOM offers the richest composition mechanism that allows to create complex containers that can extend the contained component interfaces by adding a variety of services.

	COORDINATION							
	Connector					Component Roles		
	Connector Definition		Coordination Task			Role-based Interaction		Peer-to-peer Interaction
	Implicit	First-class entity	Interface Adaption	Filtering	Ordering and Protocol	Client/Server	Master/Slave	-
Corba Component Model	implicit					client/server		event-based
SCA Component Model		bindings			several	service-oriented		
OpenCOM Component Model		bindings	allowed	allowed	allowed	client/server		
ROS Component Model	implicit					services		publisher/subscriber via topics
OROCOS Component Model		connection policies			buffering	operations		data-flow via ports

Figure 4.4: Coordination summary table.

When dealing with the coordination concern, domain specific models, in this case ROS and OROCOS, foster a peer-to-peer interaction among components. A separation of roles (client and server) is still possible although the main interaction scheme is peer-to-peer oriented. At the same time, in these component models the connector concept, if explicit, is not associated to strong coordination tasks but, as for OROCOS connection policies, it only provides means for data buffering. On the other hand, general purpose component models foster a client/server role based interaction style supported by the presence of explicit connector constructs that handle and order component interactions. In this sense, OpenCOM appears to be more similar to general purpose models. In Figure 4.4 coordination features are summarized.

SCA and ROS Integration

In this chapter we describe the integration between the ROS framework and SCA. The aim of the integration between two quite different component models is to combine the robotic-oriented features and the provided toolset of the ROS infrastructure with the flexibility offered by the SCA environment.

In the classification presented in Chapter 4 we found that component models which are focused on different application domain tend to give greater emphasis on some feature while neglecting others. In OROCOS, for example, great attention is put on the periodic execution of components while the support for multiple programming languages and the possibility of composing components into higher order structures is lacking.

On the other hand, general-purpose component models allow a more flexible building of software systems where components can be easily collected into higher-order structures, developed with different programming languages, deployed on different nodes and the interconnection among them are more flexible. These component models, such as the Corba Component Model, lack in giving some of the features that can be determinants when developing robotic software, for example the possibility of exploiting data-flow communication or imposing an execution frequency for periodic components.

It is anyhow clear that a component model that fits perfectly each need cannot exist. The particular nature of robotic software exacerbates the problem since, this kind of software applications, are usually composed by

low-level and hardware related pieces of software that need to cooperate with higher-order memory and computation intensive functionality in order to carry out the given control task.

Another critical concerns regards the hardware variability that is present in almost all robotic systems as we described in Chapter 1. General purpose component models usually assume that computational nodes involved in a distributed system are standard PCs with sufficient computational capabilities and a large bandwidth network connection. This is not true for robotics since, in particular for mobile robotics, the payload and current consumption constraints force the developers to choose lightweight and resource-poor hardware systems, at least for what regards the on-board computational units installed on the robot.

The increasing capability of the nowadays hardware devices allows to implement PC-based control systems in a range of applications in which the use of specific, and often exotic, hardware was mandatory until a few years ago. However, the continuous growth of hardware capability and the progressive reduction of its dimensions and power consumption is compensate by the increasing complexity of robot software in terms of more complex functionality and consequent computational power requirements. This leads to divide the functionality of the system on a number of cooperating networked units ranging from industrial or embedded low power PCs to remote high-powered machines.

In this context, the features offered by the component model are critical since it should be lightweight for being effectively used on on-board-resource constrained machines and, at the same time, it should allow the developers to fully exploit the computational resources of the remote nodes.

For this reason we explored the possibility of *integrating* different component models in order to exploit the robotic-oriented functionality of a domain specific component model and, at the same time, benefit from the flexibility offered by a general purpose model.

For achieving a good flexibility together with robotic specific features we choose to integrate the Robot Operating System with the Service Component Architecture. In particular we intend to exploit the Java implementation of

ROS, called *rosjava* [12], that allow to develop ROS nodes as Java applications. For what concerns SCA we adopt the Apache Tuscany environment that is natively implemented in pure Java language.

In our opinion, the use of the Java language itself gives a greater flexibility in developing applications since it allows to take advantage of the large collection of libraries and tools expressly developed and distributed for Java-based software. The Java language also offers the possibility to write fully portable code.

As we will discuss later in this chapter, the C++ programming language will be adopted only for computation critical and low-level components that need to directly interact with the robot hardware. A working example of this integration will be presented in Chapter 6.

In Section 5.1 we give a more detailed description of the Service Component Architecture and its major run-time environments. In Section 5.1 we describe the Robot Operating System giving a more detailed description of the filesystem organization and run-time structure. Finally, in Section 5.3, we analyze the implementation details of the SCA-ROS integration.

5.1 The Service Component Architecture

In this section we introduce the Service Component Architecture [14] as it is a part of the proposed SCA-ROS integration.

Since the SCA technology has been developed to address the Service Oriented Computing requirements by providing a technological service-oriented framework, the basics concepts of the service-oriented computing are introduced. The Service Component Architecture (SOC) is a paradigm for the design and the development of software that aims to evolve software systems from products to services. Following this approach, a software is considered as a set of services offered to the final users which can be humans or other software systems. The SOC approach is based on the development systems by composing heterogeneous *software services*, a software service is an element that offers reusable and self-contained functionality that can be easily integrated with other services. By heterogeneous services we intend services

that can be defined, developed and integrated using different languages and implementation technologies.

SOC is not an entirely new concept, several already existing technologies, such as CORBA, COM, J2EE and .NET can be considered, to some extent, service-oriented. A particular implementation of the SOC approach, which is receiving increasing success, are the *web services*. They are autonomous applications offering software services that can be recalled both from users and from other services by using standard formats, based on the XML, for the exchange of messages over the Internet.

The web service architecture defines four main roles: the *service provider* which is in charge of implementing and making available a service; the *service consumer*, or *client*, that invokes the service; the *service locator* which is a specific service provider that works as a registry that collect a list of available services and the *service broker* that is another type of service provider that manages the service requests from the clients forwarding them to the providers.

The increasing success of SOC lead to the definition of the *Service Oriented Architecture* (SOA) standard in 2006 [15] by the OASIS group [8]. As defined by the standard, the SOA is a methodology for the development of applications based on the composition and interaction of services through standard protocols. For example, web services define their own interfaces using WSDL [22], they can be located using the UDDI protocol [21] and they can be invoked using SOAP [16]. Services are concretely realized by means of software components structured in a *business process* that can be defined simply referring to their interactions without focusing on the underlying implementation. This composition of services is called *service orchestration* and can be expressed with specific languages such as WS-BPEL [23], XLANG² and Jolie [4].

In the service-oriented methodology, software components expose their functionality as services keeping the specific implementation well separated from the exposed interface. To this purpose, a component model is needed for defining the component construction and composition rules, however, the SOA standard provides only principles and guide lines for developing service-oriented systems without providing any specific implementation technology.

The Service Component Architecture (SCA) aims to meet those needs by defining a technology-independent and service-oriented framework. SCA is a set of specifications for the realization of distributed applications following the principles of both the service oriented computing and the component-based software engineering (CBSE). This model has been promoted by a consortium of IT industries such as IBM, Oracle, Sun and SAP and its standardization is promoted by the Open Service Oriented Architecture (OSOA) group.

The first release of SCA (version 0.9) has been published in November 2005 defining the main characteristics of a SCA system and referring to Java and C++ as implementation languages. In July 2006, SCA has been improved by adding the support for other programming languages and implementation technologies. In March 2007 the OSOA committee published the 1.0 version of SCA and several improvements are programmed to appear in the next versions of the technology.

5.1.1 Basic Principles

The software *components* are the constituent elements of SCA and they can provide interfaces by exposing *services*, require interfaces by declaring *references* and expose *properties*. Services and references are connected through connectors called *wires*. The component model has a hierarchical nature since each component can be realized by a software entity implemented in a supported programming language or it can be built by composing a number of sub-components realizing a *composite*.

The composition and the configuration of components is realized by means of a simple XML-based language that defines a SCA application called *contribution*.

The SCA design refers to four fundamental principles: *programming language independence*, *interface definition language independence*, *communication protocol independence* and *non-functional properties independence*. These principles aims to make possible the definition of a service-oriented architecture that is as far as possible independent from underlying software technologies.

- **Programming language independence:** SCA do not assumes that components are implemented in a unique programming language, mapping to different programming languages are supported instead, the most important are Java, C++, BPEL and Spring.
- **Interface definition language independence:** the components of a SCA application can provide and require functionality through interfaces that define a contract between the service provider and its clients. To this purpose, more than one interface definition language (IDL) are supported, the most used are Java and WSDL.
- **Communication protocol independence:** Generally, the most used communication modality is the web services technology, anyway, there can be cases in which this solution is not well suited. For this reason, SCA provides the *binding* notion that let the services and references to be associated to a particular communication protocol such as SOAP, Java RMI, JMS or JSON.
- **Non-functional properties independence:** A certain number of non-functional properties can be associated to a SCA component through the concept of *policy set* or *intent*. The basic idea is that a component can declare a set of non-functional features demanding to the SCA platform the task for ensuring that they comply. At the moment, in the SCA specification are included policies for security and atomicity of transactions. Moreover, because there could be different needs, the policy set can be extended by users.

5.1.2 The SCA component model

A SCA contribution is composed by one or more components whose interactions are modeled as services, this allows the developers to clearly separate the implementation technology from the provided functionality. Components can be combined and grouped into aggregate structure called composites.

A SCA composite represents an assembly of components from a logical point of view. From a functional point of view, components inside a composite

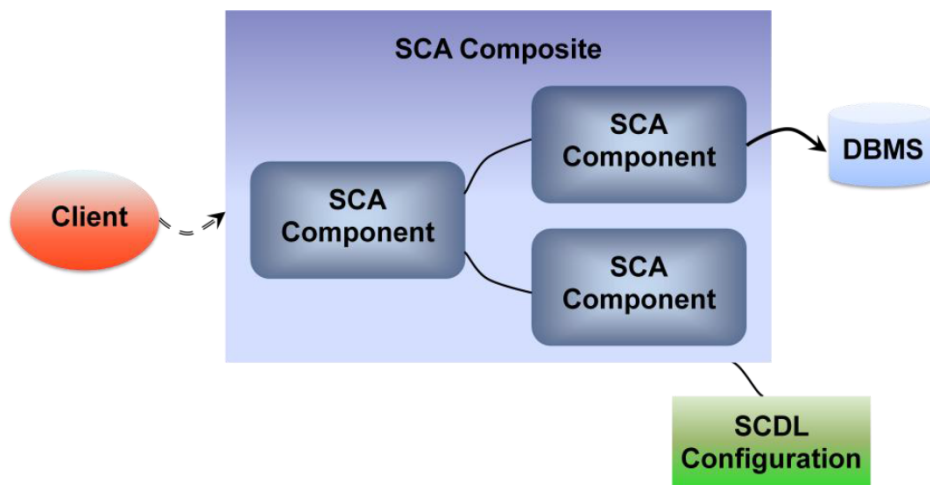


Figure 5.1: A SCA composite example.

can be executed in a single process on a single machine or they can be distributed across a set of processes executed on a network of computers. A complete application can be composed of a single composite or of a set of composites as well.

As shown in Figure 5.1 a SCA application can interact with software that do not belong to SCA such as a JSP page, a web service client or a Java class through RMI. SCA components can access data in several ways, for example through Service Data Objects (SDO), JDBC and Java Persistence API (JPA).

A SCA composite is usually described within an associated configuration file with `.composite` extension. This file uses a particular instance of the XML language called Service Component Definition Language (SCDL) that is used to describe the components, the services, the references and the wirings between them.

5.1.3 Domains

A *domain* is a specific instantiation of a SCA run-time environment executing one or more components. A domain can contain one or more composites, which, in turn, can contain more than one components executed in one or

more processes on a set of computers.

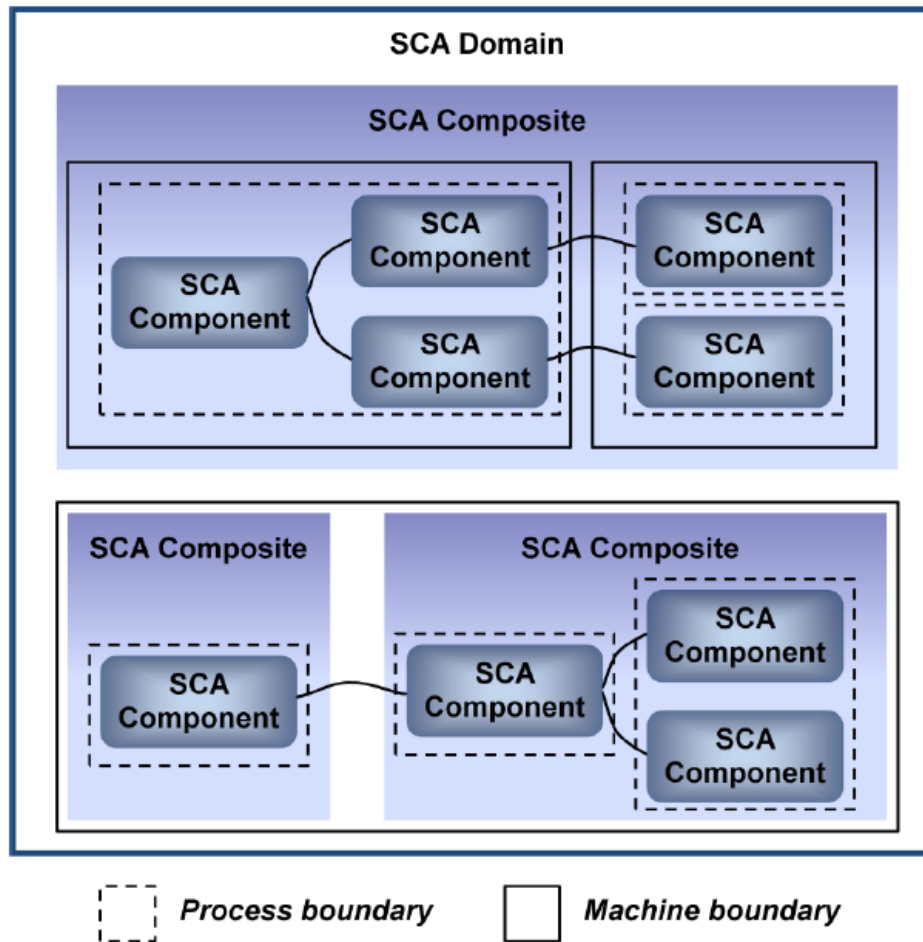


Figure 5.2: A SCA domain example.

Figure 5.2 shows an example of a SCA domain containing three composites and displaced on three computers. The first composite, in the upper part of the image, is composed of five components executed into three process distributed across two machines. The other two composites are executed by a single machine and they are divided into three separated processes. The interaction mechanisms between components are usually defined by the adopted SCA run-time environment, for example Apache Tuscany.

5.1.4 Components

Components are the base elements of a SCA contribution. Each component has a precise behaviour expressed in terms of provided and required services, hence, the functionality of the system depends on the on the configuration of its components.

In the SCA meaning, a component is an implementation instance properly configured with an XML-SCDL file defining the interaction mechanisms between the component and other components or external applications. Hence, several components can use the same implementation configuring it in different ways. Figure 5.3 shows the graphical notation adopted for the description of a SCA component.

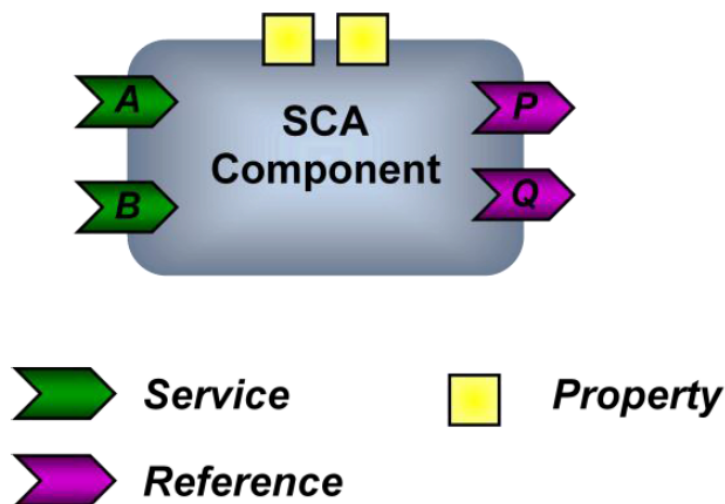


Figure 5.3: SCA graphical notation for components

SCA defines four different modes for the component instantiation:

- **stateless**: all instances of the same component are equivalent.
- **request**: a new instance is created for each request.
- **conversation**: a new instance of the component is created for each *conversation* that is requested by a client.

- **composite:** a singleton object is created for each composite.

On the basis of the selected mode, the run-time manages the creation and the initialization of new instances of components.

5.1.5 Services

Each component implements a piece of the functionality of the application and exposes one or more services represented as green arrows in figure 5.3. Services provide a certain number of operations, also called *business functions*, that can be accessed by associated clients. Generally, the modality with which services are defined depends on the technology used for the component implementation: Java components will expose services as Java interfaces while BPEL components will use WSDL to describe services.

In the SCDL file, the XML element expressing a component can have one or more children elements and attributes representing the services configuration, the principal are:

- **name:** it is a mandatory attribute corresponding to the name of the service as defined in the implementation.
- **requires:** it is an optional attribute list of non-functional features such as confidentiality, atomicity of transactions and security.
- **policySets:** it is an optional element listing additional policy sets.
- **interface:** each service can have an interface element describing the provided operations. This is a sub element of the service element. If no interface is specified, the interface specified in the service implementation is adopted. In any case, this interface, when specified, must be compatible with the implementation interface.
- **binding:** each service element can have a binding that specifies the communication protocol to be used for the interactions with the service. If no binding has been specified, the specific protocol defined in the service implementation is adopted. The communication protocol must

satisfy the policy set and the non-functional features specified for the service.

5.1.6 References

In addition to providing services to software clients, a component can also depend from services provided by other components. To describe this dependency, a component can express which services it needs through the reference concept. In Figure 5.3, references are depicted with a purple arrow exiting from the component frame. Differently from the concept of service, the concept of reference can be used to explicitly and formally show the dependency of a component from others helping the developers to clearly express the functional relationships between the components of the system.

The XML-SCDL element that represent a service can have the following attributes:

- **name**: it is the mandatory name of the reference, it must correspond to the reference name defined in the implementation.
- **autowire**: it can optionally specify if the reference should be automatically connected to the corresponding reference.
- **requires**: it is an optional attribute defining an additional list of policies and non-functional features for the reference.
- **policySets**: it is an optional attribute listing a set of additional policies.
- **multiplicity**: optionally, it is possible to define the number of possible connection between the current reference and the available services. The default value is one.
- **target**: it is an optional list of services addresses (the number depends on the multiplicity) linking the reference to one or more service provided by another component. The form is `NameOfComponent/NameOfService`.
- **wiredByImpl**: optionally, this boolean value can specify whether the implementation of component should automatically wire the reference

at run-time. If the value is `false` it means that the wire that links the reference to the service is defined *before* the execution, if `true`, the link is left undefined waiting for a dynamic match that can be provided, for example, by a service broker.

An interface may be optionally associated to a reference with the purpose of describing the required operations, this can be done in the SCDL file by defining the element *interface* as a child of the reference element. The defined interface must agree with the implementation interface which, if no interface is specified, is used by default.

5.1.7 Properties

A component can specify one or more properties. Each property is read from the SCDL configuration file when the implementation is instantiated. The properties allow to parametrize the behaviour of the component avoiding the need of making modifications on the component's implementation code.

In the configuration file, the XML element `property` can have the following attributes:

- **name**: represents the name of the property as used by the implementation.
- **type**: is the type of the property expressed by a XML schema type.
- **many**: an optional boolean value expressing whether the property is multiple or not. Multiple properties are a *collection* of values.
- **mustSupply**: if true, the property value must be supplied in the SCDL file, if false, a default value defined in the implementation can be used.

5.1.8 Bindings

Service and reference concepts define the interactions between components in a SCA system without specifying the communication mode. This is very important since it enhances the reuse of components keeping the implementation

separated from the communication protocols that are typically application dependent. For the definition of communication aspects, the SCA technology provides the *binding* concept.

A binding specifies the modality through which components can communicate with each other or with external applications. Depending on the recipient of the communication, a component could, or could not, need for a specific binding. When the communication is carried out between components being executed on different networked machines, the SCA run-time can determine the appropriate binding to be used. Moreover, services and references can have multiple bindings allowing several interaction modality with them.

The binding concept represent an important step in decoupling component computation from communication issues improving the software reuse according to the SOC principles.

5.1.9 Composite

The goal of composites, in the SCA specification, is to group components in proper schemes that can, in turn, be further composed obtaining a hierarchical structure with different levels of abstraction. This approach offers several advantages: first of all, the presence of different levels of abstraction helps the developers in the design phase of the overall system; moreover, since composite can be deployed and executed in several ways (one or more processes on one or more machines), it allows the deployment of the application as an unique abstract entity. Finally, there is the possibility to use graphical tools for the assembly, the wiring and configuration of components regardless of any implementation detail.

5.1.10 Wires

Once references and services has been defined, there is the need to specify, at the composite configuration level, which are the connections, called *wires*, that link a service to a reference. This is important since, in the same composite, there could be more than one service with a given interface and an automatic configuration may be ambiguous or impossible.

Wires can directly connect services to references in the same composite, they specify the actors involved in the communication without specifying the modality (in terms of used protocols) of the communication which is, instead, defined by the bindings. Wires are also responsible for the *promotion* of component services (or references) to composite services (or references). When a service or a reference is promoted to the composite level, it can be reached from other composites or other non-SCA software applications.

In the SCDL file, wires are defined by using the `wire` element.

5.1.11 SCA Run-time Environments

As already explained, the SCA specifications do not take into account the implementation details of the technology. This is for the purpose of ensuring the highest level of portability across different run-time environments. For this reason, several SCA platform implementation are available both in open-source and in commercial versions. Most common open-source implementations are Apache Tuscany (Section 5.1.11), OW2 FraSCAti (Section 5.1.11) and Fabric³ (Section 5.1.11). The most spread used proprietary implementation is IBM WebSphere [3].

Even if it is not binding, almost all run-time implementation of SCA are structured as shown in Figure 5.4. Each SCA run-time should provide a certain number of containers, one for each technology supported by the implementation. The *implementation* element in the SCDL configuration file notifies to the run-time which container is necessary for the execution of each single component. In the example, three components are required for Java, BPEL and Spring. Moreover, the run-time defines the communication mechanisms among components independently from their implementation technology.

The interface between the run-time and the containers is public in many SCA run-time implementations, allowing a run-time developers to extend the support for other implementation technologies and programming languages.

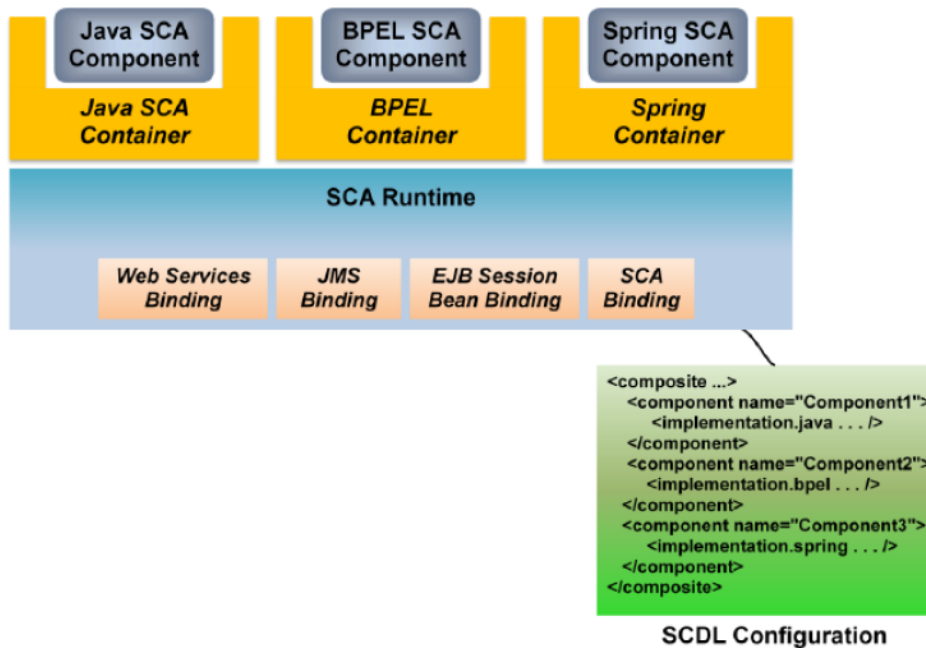


Figure 5.4: Structure of a SCA run-time.

The Apache Tuscany Environment

Tuscany [60] [1] is an open-source run-time environment developed by the *Apache Software Foundation* that provides an infrastructure for the development and management of SCA systems. Tuscany is integrated with a large number of languages for component definition, such as C++, Java, BPEL and Javascript, and a great variety of communication mechanisms.

The Tuscany architecture is focused on a lightweight implementation that allows its execution both as a standalone system or in association with other application servers like Apache Tomcat. The entire implementation is modular, this helps the integration with other technologies and the extendability of the platform. The run-time environment is implemented both in a pure Java version and in a C++ version called native.

Figure 5.5 shows the Apache Tuscany architecture that can be divided into a principal architecture and a set of extensions for other technologies. Although this makes it possible to extend the Tuscany support for other

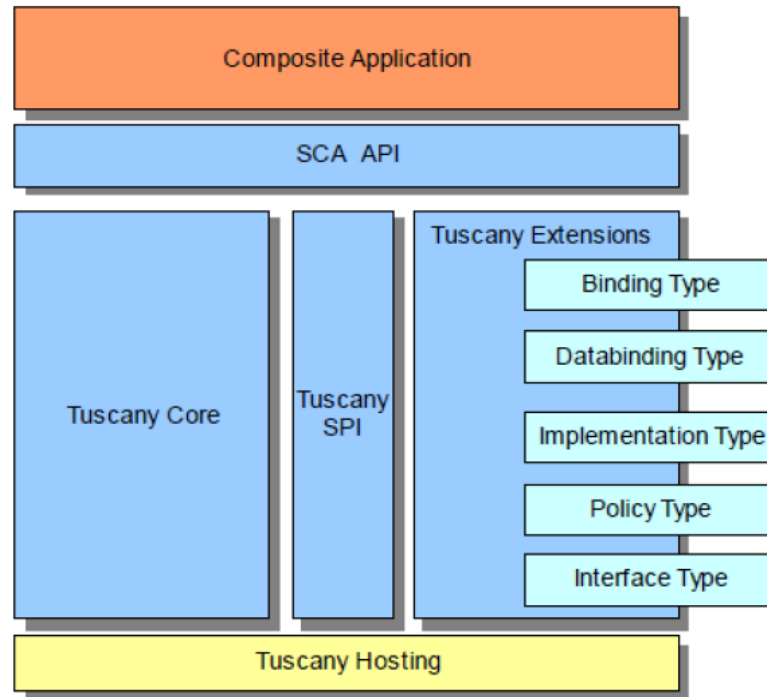


Figure 5.5: The Apache Tuscany run-time architecture.

technologies, many of them are supported by the default release of the run-time environment.

The first layer refers to *composite application* that is in charge of containing and give support for the execution of components and composites as defined in the XML-SCDL composite file. The second layer refers to the *SCA API* and provides a communication interface between component implementation and the run-time system. The APIs are specific for each supported component implementation language.

On the left part of the Figure 5.5 the *Tuscany core* is shown. This part of the run-time system is in charge of supporting components and services creation, the overall application building and their management. In the middle, the *Tuscany SPI* offers an interface that makes possible the extension of the Tuscany functionality by interfacing modules, both custom and default, with the application core. Finally, on the right, the possible extensions

are shown. Examples of extensions are: communication protocols (binding types), exchange of data formats (data-binding types), implementation types, interface definition languages and policy sets.

The bindings, in particular, provide support for a number of communication protocols like SOAP, HTTP, RMI and JSONRPC. The data-binding framework allows a transparent exchange of data between services freeing the developers from the burden of defining explicit data conversions. The implementation types extension provides support for different languages, up to now, the supported languages are: Java, BPEL, Spring and JavaScript. The interface definition language extension allows the definition of interfaces by means of different specific languages although, up to now, Tuscany supports only WSDL and Java.

The FraSCAti Environment

OW2 FraSCAti [72] [73] [9] is an interesting, although less spread used, run-time environment for SCA. It is characterized by the introduction of several innovations to the SCA standard approach. This platform integrates SCA with a set of functionality for the dynamic management of application extending the component model of SCA. FraSCAti provides a mechanisms for the handling of events among components through which it is possible to dynamically manage non-functional features like transaction control.

The FraSCAti approach is aimed by the will to implement the SOA principles in a more complete way, above all for what it is concerned with the dynamic configuration and reconfiguration of components. This aspect is often neglected by other run-time environments such as Tuscany. For this reason the FraSCAti architecture is based on the *container* concept. Containers are wrappers for components that offer several additional services which are not present in the SCA specifications.

The container is a component meta-level that provides access to various services. Each service handles a particular facet of the management of a SCA component. The result is a two-level architecture where each SCA component is hosted by a container which is itself built in a component-based fashion.

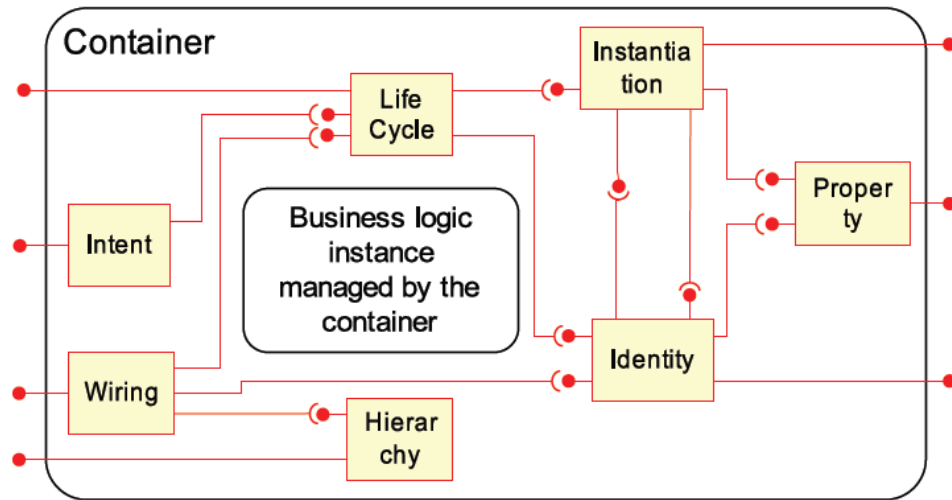


Figure 5.6: The structure of a FraSCAti container.

Figure 5.6 shows the structure of a FraSCAti container wrapping a component. The services provided by containers are:

- **Component wiring.** This service provides the ability, for each component, to query the list of existing wires, to register new wires and to remove wires. These operations can be performed on a running SCA application.
- **Component instantiation.** SCA specification define four modes for the instantiation of a component: *stateless*, *request*, *conversation* and *composite*.
- **Component property.** This service provide means for component properties set and retrieval.
- **Component identity.** Helps the management of components by querying the list of provided services and required references.
- **Component hierarchy management.** The SCA component model is hierarchical in the sense that a component can be either primitive or composite. Composite components contain sub-components which

can be, in turn, either primitive or composite. The resulting hierarchy structure is a tree. The management of this hierarchy is performed by two sub-services: one for adding, querying and removing the sub-components of a composite, and one for retrieving the parent of a component.

- **Component lifecycle.** When dealing with multi-threaded applications, reconfiguration operations, such as the ones mentioned in the previous items, cannot be performed in an uncontrolled way. Indeed, modifying a wire while a client request is being served may lead to inconsistencies and wrong results or errors returned to clients. For this reason, the lifecycle service ensures that reconfiguration operations are performed safely and consistently in insulation with client requests. This service controls the lifecycle of the components in the sense that it strictly delimits the time intervals during which reconfiguration operations can be performed and those during which application level requests can be processed.
- **Component intent.** This service manages the non-functional properties and the policies associated to each component.

FraSCAti uses *interceptors* objects to integrate services with application-specific business code keeping these concerns separated both at design time and at run-time. Each interceptor has a registered policy for data exchange (request for authentication, use of a cipher, certificates, ACID transactions and so on). Interceptors are placed by the SCA infrastructure between services and references which are annotated with the SCA annotations. When a client request is served by the corresponding server component, the request is first “trapped” and handled by the interceptors that applies the predefined corresponding logic and then the request is transmitted to the component. Interceptors act as filters and they are dynamic since they can be added or removed at run-time.

The FraSCAti platform is implemented in pure Java and uses the same component model of the FraSCAti SCA applications. The platform has four

main components: *component factory*, *wiring & binding factory*, *middleware services* and *assembly factory*.

The component factory is in charge of creating containers and components. The wiring & binding factory is in charge of creating wires between components, since SCA is independent from communication protocols, components are first specified and then they are bound using a selected distribution technology. This allows to decouple the task of designing the business logic of the component from the details of the protocol which implements the communications. Besides creating internal wires in an SCA application, the wiring & binding factory is also in charge of exporting services and references to the composite interface. The middleware services is a repository for the non-functional services which are made available to the applications. The assembly factory is responsible for parsing the SCA assembly language descriptors, interpreting the XML tags and creating the corresponding component assemblies. Whenever necessary, the assembly factory relies on the component factory for creating components, on the wiring & binding factory for creating wires, exporting services and references, and on the middleware services for integrating non-functional features into applications.

The Fabric³ Environment

The Fabric³ run-time environment [2] is officially compliant to the SCA 1.1 specifications and offers some interesting and innovative features extending the SCA concepts. The run-time can be executed as a standalone application providing a component “deployer” that allows the execution of components. Components can be put into execution and removed dynamically. Moreover, the Fabric³ run-time can run on Apache Tomcat application servers or on a Java EE application server.

When compared to other SCA run-times, Fabric³ offers two main additional features:

- **Event-based communication:** Along the common definition of services and references, components can also define communication channels allowing for an event-based communication mechanism. A data-producer

component can write new data on the communication channel and all the subscribed components can be notified about the presence of new data. Each subscriber component implements a *callback* method that is called by the run-time platform when new data is available on the communication channel

- **Periodic execution:** Fabric³ allows the definition of periodic components. This can be done by expressing their implementation in the SCDL file using a particular XML element introduced by Fabric³ (`implementation.timer`). The run-time is in charge of periodically executing the timer components at the given frequency.

However, the Fabric³ run-time has strong limitations regarding the use of periodic components and event-based communication mechanisms. First, periodic components cannot be event consumers and, hence, they cannot be notified by the run-time when new data is available on the subscribed communication channel. Second, timer components cannot be the targets of a service-to-reference wiring. This strongly limits their usability in SCA applications.

Moreover, the periodic execution of timer components is not guaranteed in any way, hence, their real execution frequency is deeply dependent from system load, complexity of computation and network bandwidth. This leads to a non sufficient accuracy in the periodic execution in most time-driven applications, especially when execution periods shorter than one second are required.

5.2 The Robot Operating System

The Robot Operating System (ROS) [66] is an open-source meta-operating system for robots developed by Willowgarage [20]. ROS is not an operating system in traditional sense of management and scheduling of processes, it rather provides a communication layer *above* the host operating system supporting the the execution of components in a distributed and heterogeneous system.

ROS is strongly focused on the robotics field and it provides means for hardware abstraction, message passing, package management, tools and libraries for most commonly used functionality in the robotics domain. The ROS framework has been developed keeping in mind five main goals: *peer-to-peer* relationship between nodes, *tool-based* support, *multi language* compatibility, *lightweight* implementation and free and open-source license.

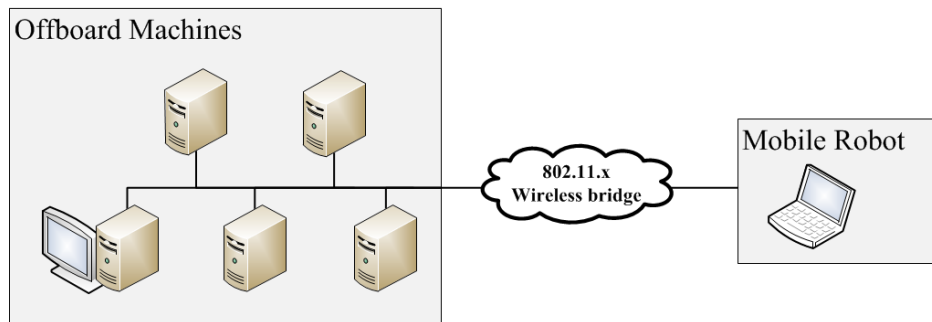


Figure 5.7: A typical ROS network configuration.

A typical configuration of a mobile robotic system is composed by a set of networked machines. One or more of them are located on the robot while others are remote and can be accessed through the network as shown in Figure 5.7. The onboard computers are usually low-powered because of current consumption, space and weight limitations on the mobile robot while the remote machines are high-powered and usually run computation-intensive tasks such as computer vision and Simultaneous Localization And Mapping (SLAM) algorithms.

A ROS system consists of a number of processes connected at run-time in a peer-to-peer topology with the aim of avoiding the presence of a central application server that could become a bottleneck for performance. To support the cross-language development, ROS uses a simple, language-neutral Interface Definition Language (IDL) to describe messages that are then autonomously translated into programming language objects by the framework. ROS is developed as a set of small tools that can be used to build and run ROS components, this approach resembles the microkernel approach for the construction of operating systems. The reusability of implemented components

is enforced by keeping the ROS implementation thin, this, according to its creators, should make the extraction of code easier. The ROS implementation is free and open-source and the framework is distributed under the terms of the BSD license which allows the development of both commercial and non commercial software. ROS currently runs only on Unix-based systems, above all Linux and Mac OS X. The ROS core system, default tools and libraries are developed and distributed into ROS distributions that are regularly released.

5.2.1 The ROS Component Model

The Robot Operating System can be analyzed according to two levels of abstraction: the *filesystem level* and the *computation graph level*. This leads to a distinction between how software entities are created, stored and managed in the host filesystem and the run-time nature of these software entities.

According to this distinction, ROS defines two types of names: *package resource names* and *graph resource names*.

ROS Filesystem level

The filesystem level concepts represent ROS resources that can be distinguished on the filesystem of the host operating system, they are: *packages*, *manifests*, *stacks*, *stacks manifests*, *message types* and *service types*.

ROS software is mainly organized into **packages** and they represent the basic unit of compilation of a ROS system. A package can contain one or more component implementations (*nodes*), a ROS independent library, a record of data, a set of configuration files or external software. In general, the content of a package, constitutes a self contained module that is useful for some task. The goal of the use and the definition of packages is to provide a flexible and easy-to-use container for reusable functionality, for this reason, packages can be created, moved and compiled using the package management tools provided by ROS.

Manifests provide meta-data describing the packages, each package must have a manifest file describing it. As a matter of fact, the minimal ROS package is composed by a filesystem folder with a manifest file associated

to it. Manifests are defined in XML language and, in addition to declaring the package name, the author and the version information, they define the dependencies of the package in terms of external resources or libraries needed both for compilation and for execution purposes. The dependencies definition is language and operating system independent and it is used by ROS command-line compilation tools to resolve software dependencies. This greatly simplifies the distribution and the compilation of ROS software across system with different configurations.

Stacks are organized collections of packages providing complete functionality. An example of stack is the *navigation stack* that provides functionality for the navigation of mobile robots in a three-dimensional environment. In other cases, libraries are distributed as stacks, for example the Point Cloud Library (PCL) that provides instruments for creating and managing three-dimensional point clouds.

Stacks are the top level hierarchy entities in a ROS system since they cannot contain other stacks. They represent the primary mechanism for distributing code being it a library, a tool or an executable set of nodes. It must be said that, in recent versions of the ROS framework, the focus has been moved from stacks to packages and, up to now, packages are often distributed individually. Similarly to the packages, stacks are described by **stack manifests** through which stack information and dependencies can be obtained both by users and by command-line compilation tools.

ROS messages are described by means of **message types** files defined by using a message description language. Each ROS message is a data structure containing both primitive types (integers, floating point numbers, strings), arrays of primitive types and/or other messages. Message definition files are stored in packages and can be added as dependencies to packages and stacks. At the compile time, message definition files are translated into executable code and automatically provide means for message serialization and de-serialization in a way that is completely transparent to the developers.

Services are defined by means of a simplified service description language encoding the **service type**. Services enable the request/response communication between nodes. The service description files are stored inside a

sub-directory of a package. They are essentially similar to messages except that they always define a couple of messages, one for the request and one for the response.

ROS Computation Graph level

In ROS a *computation graph* is a peer-to-peer network of cooperating ROS components, or *nodes*. The basic concept related to the computation graph are: *nodes*, *messages*, *topics*, *services*, *bags*, the *parameter server* and the *master*. All these concept are implemented and distributed in an unique ROS stack (`ros_comm`).

Nodes are running ROS components and they represent the minimal run-time units in a ROS system. Each node is a process that perform computation and can operate periodically or in an event-driven way or both. In ROS, robot control systems are built by composing several nodes, each one of them is in charge of managing a particular aspect of the control system, for example: getting data from a laser scanner, control the wheel motors, perform localization and planning and so forth.

All Running nodes have a graph source name that uniquely identifies them in the system. Each node has also a package resource name composed by the package name and by the executable name; this is needed by the ROS run-time to find the current executable name in the list of system packages. Hence, in ROS, nodes are simply identified by their names that must be unique.

A ROS node is written in one of the available programming languages with the use of a ROS client library such as *roscpp* per C++ nodes and *rospy* for Python nodes. The use of Lua, LISP and Java programming language for the definition of nodes is also possible.

The ROS framework encourages the division of functionality on several different nodes in order to achieve a finer-grained modularity. In the author's aims this should provide an increased fault tolerance since malfunction are insulated to individual nodes. Spreading the functionality across many nodes also reduces the code complexity in comparison to monolithic systems since

implementation details are hidden and nodes expose a minimal API to the rest of the components. By doing so it becomes easier to provide alternate implementations for nodes that can be easily substituted without the need of propagating modifications to other nodes.

The **Message** concept represents the basis for the asynchronous message-based communication of a ROS system. A message is a data structure with typed fields. ROS messages are defined by means of a message description language that allows the definition of primitive types, arrays of primitive types and nested definition of other messages. Referring to this feature, ROS messages are similar to C structures.

ROS client libraries (such as *roscpp* and *rospy*) implement message generators that translate message definition files into the target source code. The source code translation of messages automatically creates serialization and de-serialization functions for message delivery.

A ROS message can include a special data type called *header* containing common data fields such as the timestamp, an integer number describing the time instant to which the message refers, the frame ID, that represent the geometric frame to which the message is related, and the sequence number. The presence of this header is strongly related to the main objective of ROS: developing robot control systems. Having time and geometric information attached to each message simplifies the handling of timed sequences of data coming from several sensors. When dealing with laser scanners, for example, the frame ID field stores the geometrical frame information that should be associated with the data and the timestamp field correspond to the time at which the scan was taken.

Messages can be exchanged via a transport system with a publisher/-subscriber semantics. Nodes can publish messages to **topics** and retrieve messages by subscribing to other topics. Each topic is uniquely identified by a string name. There may be multiple concurrent nodes publishing on the same topic and multiple nodes subscribed to the same topic. Moreover, a single node can publish and subscribe to multiple nodes. In general, publishers and subscribers are not aware of each others' existence.

The underlying principle is to fully decouple the production of information

from its consumption decreasing the space and time coupling between nodes. The only constraint related to the publishing and subscribing operations lies in the message types: each topic has a supported type and, hence, any node can publish on a topic as long as messages has compatible types.

The communication is event-based, the client library provides the definition of a callback function that can be specialized and associated to a subscribed topic. This function is asynchronously called when a new message is available on the subscribed topic allowing the subscriber node to retrieve the message content.

Topics are intended for unidirectional streaming communication, moreover, ROS allows the request/response semantics for communication by providing the service concept.

Topics are implemented as TCP/IP-based or UDP-based transport systems. The default transport system uses TCP/IP sockets and it is known as *TCPROS*. It streams data over persistent TCP/IP lossless connections. Since *TCPROS* is the default data transport, all ROS client libraries have to support it. The UDP/IP transport, known as *UDPROS* is a lossy low-latency transport supported only by the *roscpp* client library. Using an IP based communication system, the message transfer between local or remote nodes is fully transparent and the same node can be executed and reached either locally or remotely. Local and remotes computation systems running ROS nodes only need to lie on a commonly addressable network segment.

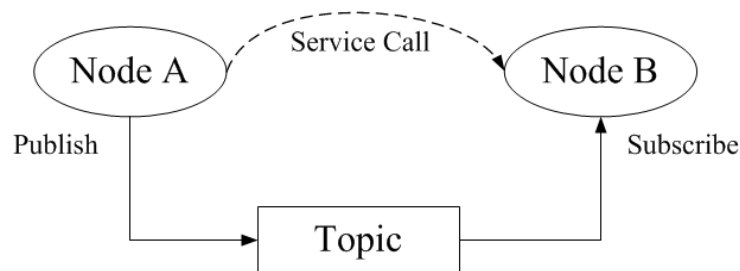


Figure 5.8: Communication between two ROS nodes.

Although the publisher/subscriber model is flexible and ensures time and space decoupling between communicating components, it is not well suited

for request/reply interactions, for this purpose, ROS provides the **service** concept.

Services are defined by a pair of message structures, one for the service request and one for the reply. A service provider node can define a service name that can be used by its clients to send request messages. Once the request has been sent, the client waits for the reply.

Client libraries present this form of interaction with the same semantics of a remote procedure call although the ROS infrastructure implements it with a message exchange over the same TCP/IP transport used for messages.

Services are defined using a simplified service definition language that indicate the request and response message formats. These definitions are then automatically translated into source code by the client library that adds the serialization and de-serialization functions as it is done for ROS messages.

Bags are data collection structures. They can subscribe to one or more topics and store the received message data into a file. This bag files can then be played back to the same topics they were received from or on a different topic.

From this point of view, the use of a bag file is no different from the presence of a ROS node sending the same data types. Bags can be very useful when testing or debugging control systems: the same data, representing for example the complete scan of a real-world environment, can be recorded once and used several times to test different mapping algorithms. Bags can also be used as advanced logging instruments collecting output from working robots.

A **parameter server** is a shared dictionary of variables and constants accessible through the network. Nodes can use this server to store and retrieve parameters at run-time, i.e. configuration data or properties. The parameter server is implemented using XMLRPC and its API are accessible through standard XMLRPC libraries by nodes. Parameters are stored with the same naming convention adopted for identifying nodes and topics in a ROS system. This leads to a hierarchical structure of parameters that can be accessed individually or as a tree.

The **Master** is the central concept of a ROS run-time system. It is a ROS node that provides naming and registration services to other nodes. It

keeps trace of publishers and subscribers to topics and services. The master allows nodes to locate other nodes enabling the peer-to-peer communication. It also provides and maintains the parameter server.

The ROS master functionality can be accessed through a XMLRPC-based API that it is used by client libraries to call and retrieve information about the graph node. However, the access to this low-level functionality is fully hidden by the client libraries and, hence, custom nodes do not need to directly access the master API.

Nodes connect to other nodes directly, the master only provides lookup information so that nodes that subscribe to a topic will request connections from nodes that publish on that particular topic. This architecture allows for decoupled operation where actors (topics or nodes) are referred by their names that must be unique in the entire ROS system. For this purpose, the naming convention adopted by ROS allows for the definition of nested names organized in a tree-like structure.

The implementation of the ROS master is provided by the `rosmaster` package.

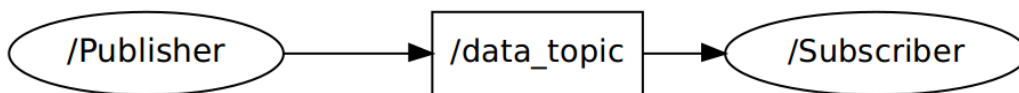


Figure 5.9: A ROS graph node.

Figure 5.9 shows a running ROS graph node. Nodes are elliptical blocks marked with their names, topics are rectangle shaped blocks and arrows show the sense of the communication. In this basic example the `“/Publisher”` node is publishing data on the `“/data_topic”` topic and the `“/Subscriber”` node is subscribed to the same topic. The image was taken using the `rxgraph` tool.

5.2.2 ROS tools

ROS is one of the most spread used framework for component-based robotics software. One of the key-feature that led to such a success is the presence of

several tools for helping the development, the debugging and the deployment of robotics applications. These tools perform various tasks, e.g., navigate the source code tree, set and get configuration parameters, visualize graph node and connection topology, logging of input and output data, graphically plot message data, generate documentation, and so on.

All these services are implemented by tools that are not part of the ROS master, everything is put into separated modules keeping the master implementation reasonably simple and efficient.

ROS provides tools for compilation and execution management of components. The `rosmake` is a compilation tool based on `CMake` that explores the dependencies defined in package and stacks manifest files and resolves the compilation dependencies between packages and external libraries. If properly configured, `rosmake` is also in charge of translating messages and services definitions into target component's source code. The `rosdep` tool works in collaboration with `rosmake` and it is in charge of finding the dependencies between the package to be compiled and other ROS packages. In the case in which some dependent packages are missing on the target system, `rosdep` can be configured to retrieve their source code from the ROS on-line repository and automatically download and compile the needed sources.

Components configurations, expressed in terms of active components and involved communication topics, can be stored in XML *launch* files. In these files it is also possible to “remap” the name of topics to other names making their reuse easier. Without the remap feature, in fact, the topic names are hard-coded inside the component implementation files and this can hamper the reusability of implementations and the flexibility of the entire system. Launch files are executed by the `roslaunch` tool that analyzes the file, loads all needed components and remaps topic names.

ROS offers a set of tools for data visualization and logging. In particular, geometric data can be visualized through the `rviz` program that allows to subscribe to topics exchanging three-dimensional robot models, laser scans, images and point clouds. `Rviz` behaves as a standard node offering a graphical user interface to the user who wants to visualize data by simply subscribing to interested topics. The `rxplot` tool offers a simplified graphical instrument

to plot and compare timed data as it could be done with MATLAB-Simulink scopes or similar tools. The `rostopic` tool allows to print data traffic from a topic into a textual UNIX-like console and allows the user to manually insert new data for debugging purpose.

To simplify and harmonize the treatment of spatial three-dimensional frames, ROS provides a geometric transformation system called `tf`. Robotic systems, in fact, often need to deal with spatial relationships between sensor and robots, between robots and fixed environment frames and between manipulators joints. For this purpose, the `tf` builds a dynamic transformation tree which relates all frames of reference of the system building a tree-like representation. The tool can be used to obtain the geometric path between a frame and another one avoiding tedious, error prone and difficult to debug manual implementation of geometric transformations.

All ROS packages are open-sourced and this greatly improves the possibility to interface ROS with other spread-used frameworks or libraries. In many cases external tools can be wrapped into ROS packages; this is the case of the OpenCV [7] computer vision library or the Player [19] software. The entire OROCOS framework [18] has been integrated into ROS and a complete working OROCOS environment can be installed in a ROS system. The ROS-OROCOS integration allows to directly connect ROS topics to OROCOS ports obtaining an hybrid system in which, typically, hard real-time tasks are carried out by OROCOS and soft real-time tasks, such as the motion planning, are managed by ROS.

The well known Gazebo robot simulator [17] has been fully integrated with ROS providing a flexible dynamic simulator for both manipulators and mobile robots. The simulation model can be interfaced with the ROS system by subscribing and publishing data from and to topics. In Figure 5.10 a screenshot taken from the `rxgraph` tool is shown.

In this case, the KUKA Youbot Arm with its gripper is simulated with the Gazebo simulator. The `gazebo` node publishes several topics including the joint states (`joint_states`) of the robot containing positions, speeds and torques of the simulated arm and the `tf` transformations representing the geometric transformations among the five joints of the robot arm.

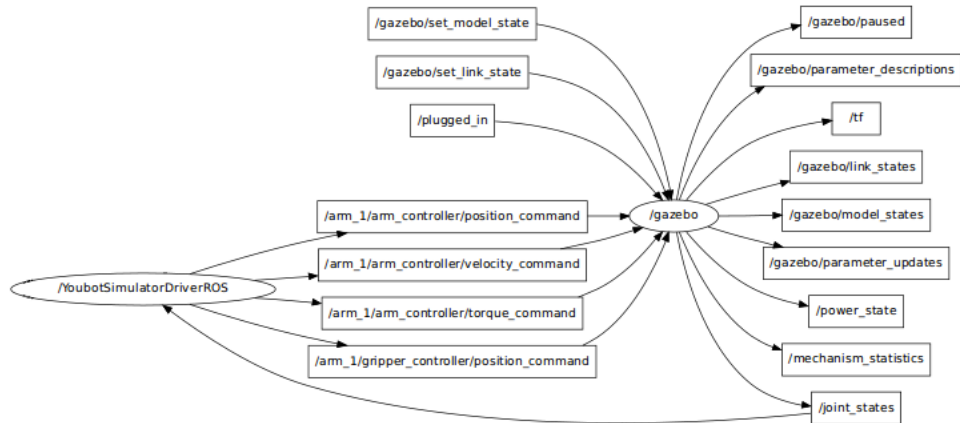


Figure 5.10: An example of a ROS-Gazebo integrated system.

The `YoubotSimulatorDriverROS` node publishes positions, velocities and torques commands on the corresponding topics which are subscribed by the simulator and reads the joint states from the already mentioned topic published by the simulator.

The kinematic and dynamic model of the robot is stored in a XML file (URDF) that can be loaded by the simulator for obtaining the dynamic and geometric features of the arm including the inertial matrix, the joints limit angles, the link dimensions and so forth.

5.3 Integration of SCA and ROS

An integrated software system comprising both SCA and ROS is composed by two almost separate subsystems, the first is composed by ROS nodes handling the low-level functionality of the robot and interfacing with the robot hardware composed by sensors and actuators. The second subsystem is composed by SCA components implementing higher-level functionality such as mapping, navigation, visual object recognition and graphical user interfaces.

The components of the SCA subsystem are implemented in Java or in any

other language which is supported by the run-time environment while ROS subsystem components will be mainly implemented in C++ and, if needed, in Java language.

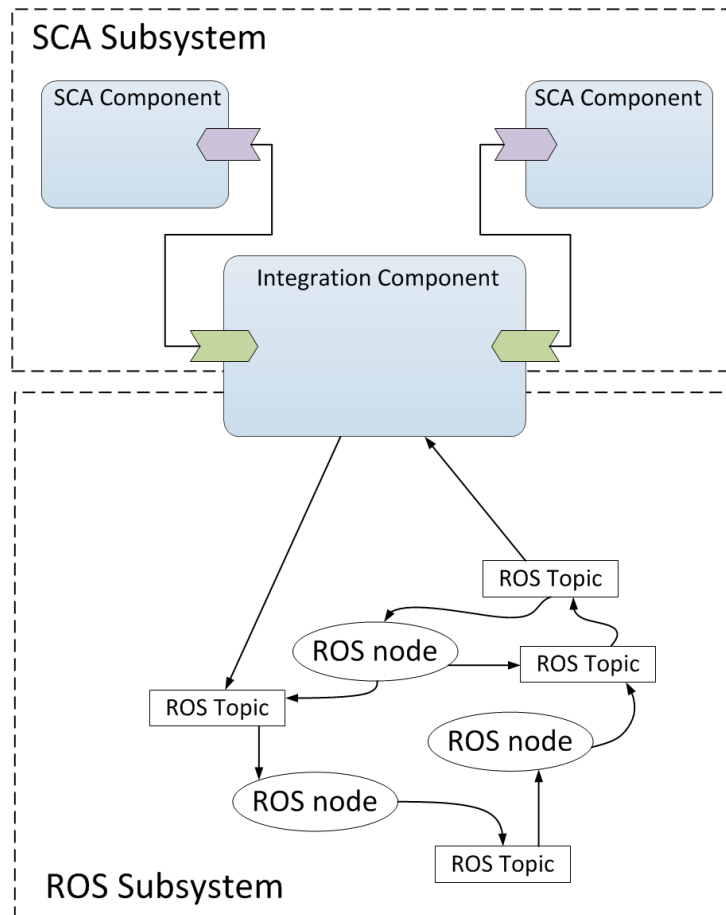


Figure 5.11: Conceptual view of the SCA-ROS integration.

The *core* of the integration process is a component that should belong both to the ROS subsystem and, at the same time, to the SCA subsystem. This is actually the approach adopted for integrating SCA and ROS in this work.

The RosGate component embodies the link between the ROS and SCA subsystems. It is implemented as a Java class that uses the *rosjava* API to interface the ROS subsystem and exposes a Java interface that can be easily

integrated into a standard SCA Java component.

5.3.1 Rosjava

As already mentioned, the `RosGate` component exploits the *rosjava* APIs to interface with the ROS subsystem. *Rosjava* is an alternative implementation of ROS written in pure Java language. It provides a client library, which behaves similarly to the standard C++ or Python ROS client libraries, that enables developers to interface with ROS topics, services and parameters. A Java native implementation of the `roscore` is also available although *rosjava* nodes can be executed and handled by the Python or C++ `roscore`.

Unlike the standard ROS client libraries, in which each node is an independent process, a certain number of *rosjava* nodes can run in a thread pool into a single Java Virtual Machine. Each *rosjava* node must implement the `NodeMain` interface shown in Listing 5.1.

- `getDefaultNodeName`: it returns the name of the current node as expected by the ROS system to build the graph node.
- `onStart`: this method represent the entry point of the node execution program. The `ConnectedNode` parameter is used to build publisher and listeners for topics.
- `onShutdown`: this is the first of the possible exit points of the node execution program. This method will be executed when the shutdown procedure is started.
- `onShutdownComplete`: it represent the final exit point of the node execution. This method is intended for managing the clean-up operations.
- `onError`: this method is called when an internal error occurs.

Publish and Subscribe operations are managed by means of dedicated objects, called `publishers` and `subscribers` respectively. Each publisher object is associated with a topic name expressed by means of a plain string and a data type. Java data types for messages can be generated automatically

by the ROS message generator starting from message definition written in the ROS Message Description Language. Similarly, each subscriber object is associated with a topic name and a data type. In addition, each subscriber can be associated to one or more listeners implementing the `MessageListener` interface. This interface provides the definition of the `onNewMessage` method that is called when a new message arrives in such a way that is similar to the callbacks of C++ ROS nodes. A ROS Java node can then manage any number of subscribed or published topics by means of a set of dedicated objects.

```
public GraphName getDefaultNodeName();
public void onStart(ConnectedNode node);
public void onShutdown(Node node);
public void onShutdownComplete(Node node);
public void onError(Node node, Throwable throwable);
```

Listing 5.1: The `NodeMain` interface.

5.3.2 The RosGate Component in ROS

The `RosGate` implementation exploits the *rosjava* APIs, figure 5.12 shows the UML class diagram of the `RosGate` component.

The implementation is organized in three main classes and one interface. The `RosGateNode` class actually implements a *rosjava* node and, besides implementing the `NodeMain` interface it also implements the methods specified by the `RosGateIntf`. The main methods are:

- `createListener(String, String):void`: this method subscribes the topic which name is specified as first argument. If the topic is not existing yet, it will be created. The second argument is a text string representing the description of the data type exchanged on the topic.
- `addObserverElement(String, Observer):void`: adds to the topic specified as first argument the observer object passed as second ar-

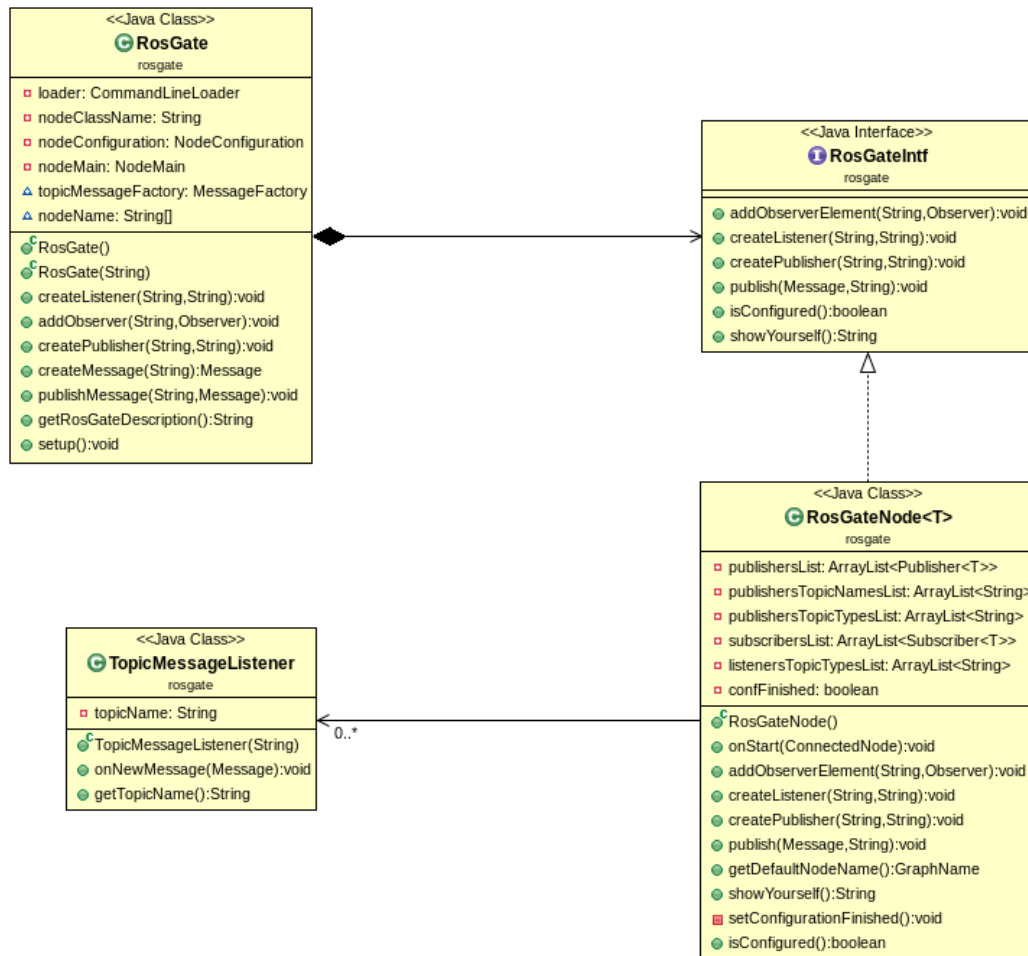


Figure 5.12: UML class diagram of the RosGate component.

gument. This observer will be notified when new data is received on the listened topic.

- `createPublisher(String, Observer):void`: this method publish the topic which name is specified as first argument. If the topic is not existing yet, it will be created. The second argument is a text string representing the description of the data type exchanged on the topic.
- `publish(Message, String):void`: this method publishes the **Message** object passed as first argument on the topic identified by the string

provided as second argument. The `Message` interface is implemented by all objects that can be sent or received via topics. Message implementations are created by the *rosjava* client library by translating the message definition files.

Other methods are available for supporting the configuration phase and for debug purpose. The `RosGateNode` implementation keeps trace of all subscribed and published topics. Subscribed topics are automatically connected to a `TopicMessageListener` objects that implements the `MessageListener` interface provided by *rosjava* APIs. These `onNewMessage` methods of these objects are automatically called by the *rosjava* run-time environment when new messages are received on the associated subscriber's topics. In the proposed implementation, this object is in charge of notifying all observer objects associated to the topic, by doing so, the new message events and the message data can be received by objects outside the ROS subsystem.

The `RosGate` class is used to mediate the access to the `RosGateNode` object. It offers the same methods for publishing and subscribing topics, it intercepts the exceptions that may be raised by the node and it manages the node creation and configuration phases. Due to limitations on the current release of the *rosjava* client library, topics publishers and subscribers must be known *before* the node execution. In order to overcome this limitation, the `RosGate` class supports two separate phases: the configuration and the run-time.

The configuration phase spans from the creation of the `RosGate` object by calling one of the provided constructors to the `setup` method call. Once the `RosGate` object has been instantiated, it looks for a running *roscore* on the network node specified by the URI passed as parameter or on the default URI if no parameters are passed. Once the connection is established, it allows its clients to add topic subscribers, associate observers and add topic publishers.

Once all publishers and subscribers have been created, the `setup` method can be called. This method creates the concrete instance of the ROS node and notifies it to the ROS system. After the setup, the class can be used as a unique access point to the ROS infrastructure and the `RosGate` node is fully integrated with the remaining parts of the underlying ROS system.

The `RosGateNode` class can raise four dedicated Java exceptions that are omitted in Figure 5.12.

- `AlreadyConfiguredNodeException`: this is raised when an attempt of adding a topic publisher, or a subscriber, or an observer is made on a node that has been already configured.
- `NotYetConfiguredException`: this exception is thrown when an attempt of publishing data is made before the end of the configuration phase.
- `TopicAlreadyExistingException`: this is raised when the client tries to create more than one publisher (or subscriber) on the same topic.
- `TopicNotExistingException`: this exception is thrown when the client tries to publish or to add an observer on a topic that it is not existing yet.

The `RosGate` class provides also a message factory mechanism that encapsulates the creation of objects representing ROS messages. By doing so the user can obtain an instance of any supported message type, being it default or user defined, by providing its string standard representation, without any knowledge about its implementation details. This functionality is available during the entire life-cycle of the class.

5.3.3 The `RosGate` Component in SCA

From the SCA point of view, the `RosGate` component is nothing more than a standard SCA component that uses the `RosGate` Java class in its implementation. The interactions with ROS are all encapsulated into the component implementation, hence, no knowledge related to ROS needs to be shared with other SCA components.

The core of the `RosGate` component in SCA is a class. The initialization method is in charge of instantiating the `RosGate` Java class, creating publisher, subscribers and adding external observers to capture the incoming data on subscribed topics.

Since no standard mechanisms supporting event-based communication are provided by the standard SCA component model, each operation related to data publishing and subscribing is made available to the remaining part of the SCA system as a service provided by the `RosGate` SCA component. For this reason, this component is viewed a server for all other components that need to interact with the ROS subsystem.

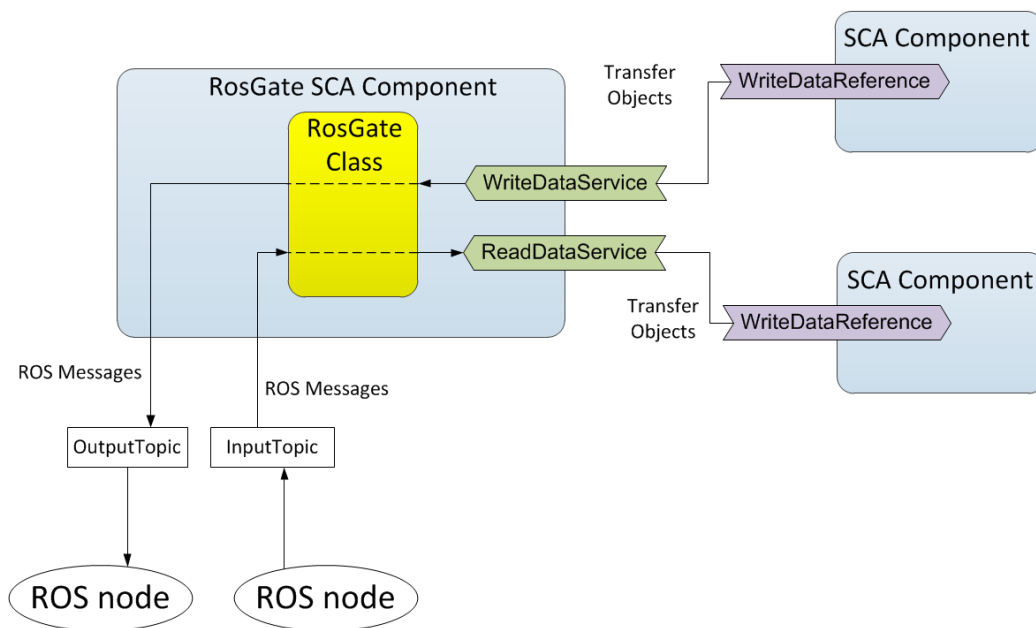


Figure 5.13: Integration between SCA and ROS systems.

To avoid the use of Java implementations of ROS messages outside the `RosGate` component, the *data transfer object* (DTO or TO) design pattern is adopted. A Data Transfer Object is an object that carries data between software actors, being them process, classes, components or other. The concrete implementation of a transfer object is a simple class that does not offer any behaviour except for the methods dedicated to the storage and retrieval of its internal data (“getter” and “setter” methods). A transfer object can be associated to each ROS message and it can be used to make the content, or payload, of ROS messages available to other components without the need to share any knowledge related to ROS. By means of this pattern,

more sophisticated approaches for data serialization and de-serialization can be implemented to exchange efficiently large data objects, such as images.

In the conceptual example shown in figure 5.13 a couple of SCA components communicate with other two ROS components, that can be developed in Java, C++ or in any other supported language and deployed on any computational node of the network.

The SCA components exchange data by means of services and references. Exchange data is encapsulated into Transfer Objects with the `RosGate`, this allows a complete independence from the ROS-dependent messages. On the other hand, the `RosGate` component exchange data with the ROS node through topics by using ROS messages. The `RosGate` class (filled in yellow in the image) provide means for publishing and subscribing data and for creating ROS messages.

The current implementation of the SCA integration component defines a property that indicates the URI of the *roscore*.

```
1 public class RosGateSCA implements SendTwist, ReceiveOdometry{
2     private RosGate rosGate;
3     private OdometryMessageObs odoObserver;
4     private MessageManager msgManager;
5
6     @Property(name="RosMasterURI", required=true)
7     protected String RosMasterURI;
8
9     @Init
10    public void init(){
11        RosGate = new Ros(RosMasterURI);
12        RosGate.createPublisher("twist_command",geometry_msgs.Twist._TYPE);
13        odoObserver = new OdometryMessageObserver(msgManager);
14        RosGate.createListener("odometry_data",nav_msgs.Odometry._TYPE);
15        RosGate.addObserver("odometry_data",odoObserver);
16        RosGate.setup();
```

Listing 5.2: The `RosGateSCA` initialization.

Listing 5.2 shows the initialization method of a `RosGateSCA` component that interacts with a generic robot driver implemented in ROS by exchanging `Odometry` and `Twist` messages. The `RosMasterURI` property identifies the URI of the current master in the system. A new `RosProxy` object is firstly created (line 11) and then used to define a published “`twist_command`” topic (line 12) and to define a subscribed topic “`odometry_data`” (line 14). A message observer is associated to the “`odometry`” topic at line 15. Listing 5.3 shows the definitions of the services offered by the `RosGateSCA` component.

```
public interface SendTwist {
    public void sendTwist(TwistTO twist);
}

public interface ReceiveOdometry {
    public OdometryTO receiveOdometry();
}
```

Listing 5.3: The `RosGateSCA` services.

Transfer objects (whose name ends with “TO”) are exploited in order to exchange ROS-independent data with the `RosGateSCA` component. The `MessageManager` object associated to the odometry message observer at line 12 is in charge of filling transfer objects with the message content and managing the message queue.

Some clarifications need to be provided about the deployment of the SCA-ROS integrated system. As we said before, ROS nodes can be deployed on a distributed system composed of several networked machines. The `RosGate` node does not make an exception to this since it can be deployed on any computation node of the ROS network. On the other hand, SCA components can be deployed on any machine given that they can access the same network address space. The only constraint added by the integration is that the `RosGate` must be deployed on a machine that lies both in the ROS and in the SCA network, since it is a ROS node and a SCA component at the same time. In real-world systems this should not represent a limitation since both

ROS and SCA networks are LAN based and, hence, all machines lie in the same network segment or in reachable networks.

The Bart Robot

The BART (Bergamo Advanced Robotic Transit) is a wheeled mobile robot developed at the Software for Experimental Robotics Laboratory (SERL)¹ of the department of Engineering of the University of Bergamo². The main goal of the robot is to provide an experimental platform to test the development of control software on a distributed system.

The robot mechanical structure is similar to a tricycle but, instead of having an actuated steering axis and a traction wheel, it has a differential drive block with two independently actuated wheels connected to the frame by means of a rotation joint. This rotation joint, that represent the steering mechanism, is fully passive since it is not actuated by a dedicated motor. The possibility of changing the steer angle is given by the on-place rotation capability of the differential drive system. A couple of fixed wheels on the rear part of the robot provide the needed support for the entire mechanical structure.

The two wheel of the differential drive block are actuated by a couple of independent DC motors with gearboxes and incremental encoders, the steer position is measured by a third rotary encoder.

The robot has three on-board computational units: two 32-bit microcontrollers and a compact embedded PC. One or more remote machines can interact with the robot by means of a wireless 802.11 connection. The first

¹<http://www.unibg.it/serl>

²<http://www.unibg.it>

microcontroller, which plays the role of slave, is placed on the differential drive block and it is in charge of controlling the wheels speeds and reading the steer rotary encoder position, the second microcontroller, that is the master, computes the kinematics relations while the PC supervise the robot functioning and gives access to the wireless network.

The microcontrollers communicate with each other by means of a CANBUS connection, the embedded PC can interact with the master by exchanging data through a serial RS-232 communication interface. The robot communicates with the other remote machines by means of the wireless LAN connection made available by the embedded PC.

In this chapter we will present the mechanical structure, the on-board electronics and computation devices, and the communication system and we will focus on the architecture of the component-based software control system.

6.1 Mechanical Structure

The robot frame is realized with a lightweight triangular structure of aluminium bars. Each side of the triangle measures approximately sixty centimeters and the steer axis of the differential-drive block is placed close to the front triangle vertex. The central part of the frame houses two shelves on which batteries and electronics boards are placed.

The differential-drive steering block is fully realized in aluminium and it houses, besides motors and encoders, the slave microcontroller and the interface boards for motors and encoders. A couple of ball bearings are mounted on the steer axis for smoother movements. The steer axis can rotate continuously, this is made possible by the presence of a slip ring collector that carries both power supply lines and Canbus lines outside the differential-drive steering block.

Figure 6.1 shows a picture of the BART robot. The driving wheels are made of PVC and have a rubber tread to increase the grip, the diameter is 100 millimeters. The rear support wheels are mounted on a spring suspension that helps in keeping a good contact between the wheels and the floor in case of roughness surfaces.

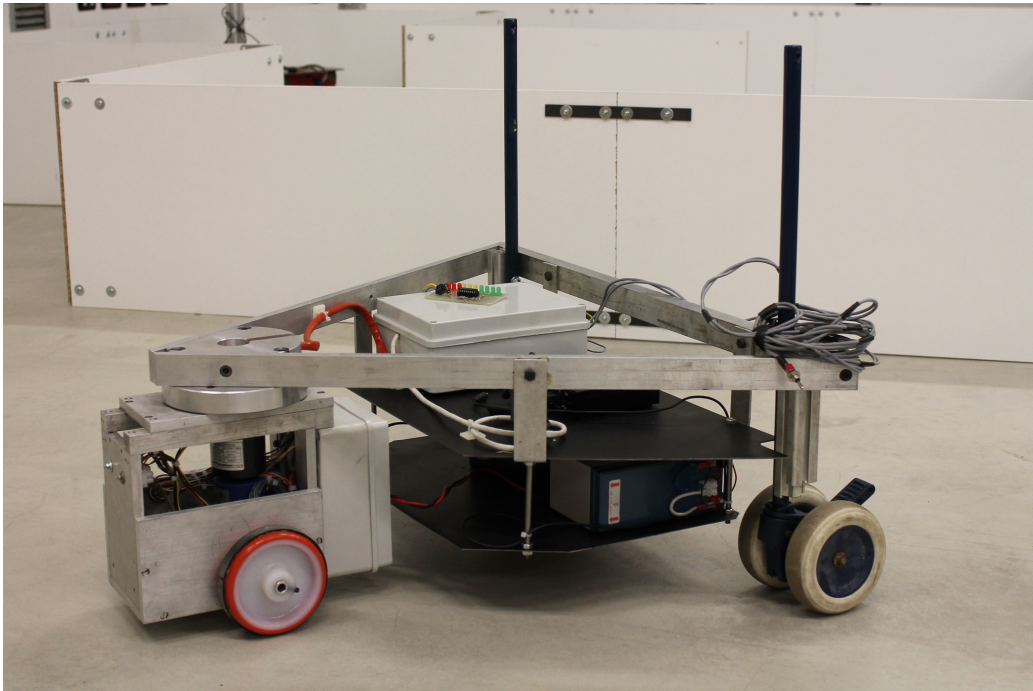


Figure 6.1: The BART robot.

6.1.1 Motors and Encoders

Each wheel is actuated by a brushed DC motor³. The nominal voltage of these motors is 12 V and the output power is 79.2 W. The weight of each motor is 242 g.

Motors are equipped with angular gearboxes⁴ with a reduction ration of 25 and an efficiency of 55%. The driving wheels are directly connected to the output shaft of the gearbox in order to reduce the backlash that would adversely affect the precision of the motion measurements.

The position of the motor shaft can be measured by means of a magnetic rotary encoder⁵ with a resolution of 512 pulses for revolution. Output data is provided by a dual channel digital output line. By reading output pulses in quadrature mode, the effective resolution reaches 2048 steps per revolution

³Faulhaber model 3257 012 CR

⁴Gysin model GSR 17

⁵Faulhaber model IE2-512

with rotation sense information. The encoders are incremental and no zero signal is provided as output.

The steer axis position is measured by means of an optical incremental rotary encoder⁶ with a resolution of 2048 pulses per revolution that leads to an effective resolution of 8192 steps per revolution due to the just mentioned quadrature reading mode. The outputs are provided by a push-pull circuitry driving a dual channel output line. The information regarding the rotation sense can be estimated by analyzing the phase between the two channels during the rotation. A third push-pull output gives a single pulse on each complete rotation of the encoder shaft. This signal can be used to find the zero position of the encoder at the robot start-up phase (homing).

6.1.2 Kinematics

The kinematic structure of the BART robot is very similar to a tricycle in which the actuated steer mechanisms is replaced by a differential drive mechanism. The robot can move along arches of circumference which centre represents the *Instantaneous Centre of Rotation* (ICR) of the robot.

The ICR is the cross point of all axes of the wheels. This point must be unique instant by instant since this condition guarantees that no translational slip occurs between the wheel and the floor.

Given the kinematic structure of the BART robot, the ICR position can lie at any point of the straight line passing through the rear wheels axes. Since the ICR must lie on this line and the rear wheels are not steerable, the position can be unambiguously expressed by the distance between the ICR and the robot centre assuming that positive values represent left handed positions and negative values represent right handed positions. For the same reason, the ICR position determines, or can be determined, by the steer position of the robot.

According to the ICR position, the robot can only rotate around that point drawing a circumference. Two notable configurations can be identified: if the ICR is overlaid on the robot centre, and hence the steer angle is 90°

⁶Eltra model EL50FA

and the ICR radius is null, the robot will rotate on place. Instead, if the ICR is infinitely far from robot, and hence the steer angle is null and the ICR radius tends to infinity, the robot will move straight. In all other cases, the robot will rotate along a circumference at a given angular speed ω as shown in Figure 6.2.

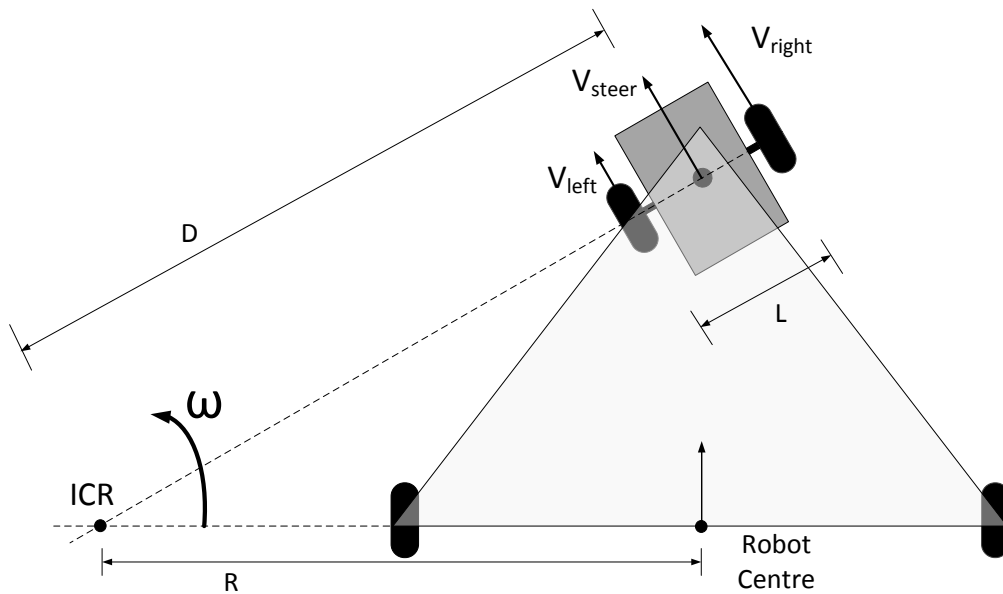


Figure 6.2: The kinematics scheme of the BART robot.

The robot can be driven by providing the ICR position and the desired angular speed ω or by providing a desired relative forward speed V and a desired angular speed ω . We will refer to this kind of command as *twist* later on in this chapter. This couple of values (V, ω) will unambiguously define an ICR radius R since:

$$R = \frac{V}{\omega}$$

Once the ICR position has been determined, the target speed of the wheels in order to achieve the target rotation can be computed in a way that is similar

to a standard differential-drive robot, in fact, given D the distance between the differential-drive centre and the ICR and L the wheel axis distance:

$$V_{right} = \omega \cdot \left(D + \frac{L}{2} \right)$$

$$V_{left} = \omega \cdot \left(D - \frac{L}{2} \right)$$

The relation between the V_{steer} , the input ω and the speeds:

$$\omega = \frac{V_{right} - V_{left}}{L}$$

$$V_{steer} = \frac{V_{right} + V_{left}}{2}$$

The D distance is equals to the ratio between the V_{steer} and the ω speed, so:

$$D = \frac{L}{2} \cdot \frac{V_{right} + V_{left}}{V_{right} - V_{left}}$$

Given the the wheel radius r the V_{left} and V_{right} speeds can be expressed in terms of angular speed of the gearbox output shaft:

$$V_{right} = \omega_{right} \cdot r \quad V_{left} = \omega_{left} \cdot r$$

Hence, given a couple of speeds V and ω , it is possible, after having found the distance D between the steer axis and the ICR, compute the target speeds to drive correctly the robot.

Now assume that the robot steer centre is at some position (x, y) , headed in a direction making an angle θ with the X axis of the environment absolute reference system. By manipulating ω_{right} and ω_{left} we can get the steer centre point to move to different positions and orientations. The ICR position can be computed as:

$$(ICR_x, ICR_y) = [x - D\sin(\theta), y + D\cos(\theta)]$$

At time $t + \delta t$ the steer centre pose will be:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega\delta t) & -\sin(\omega\delta t) & 0 \\ \sin(\omega\delta t) & \cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICR_x \\ y - ICR_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICR_x \\ ICR_y \\ \omega\delta t \end{bmatrix}$$

Accordingly, the the robot centre position can be easily found by applying the geometric rigid transformation that links the steer centre and the robot centre. This transformation is determined by the geometrical dimensions of the robot and by the angle between the steer centre axis and the robot frame; this angle is exactly the same measured by the steer encoder. This pose estimation procedure is involved in the odometry computation.

6.2 Electronics and Computation devices

The entire robot is power supplied by lead acid rechargeable batteries providing up to 28 Ah at a nominal voltage of 12 V. In standard conditions this is sufficient for several hours of robot functioning. Normally the robot is fitted out with only two batteries that guarantee a couple of hours of functioning. The robot is equipped with several electronic boards including for power supply, motors management, encoder data acquisition and microcontrollers boards.

The motors are actuated by a Pololu MD03A⁷ board based on two H-Bridge motor drivers⁸ that can deliver up to 30 A to each motor. The control is in pulse width modulation (PWM) for each motor, one input line is exploited for providing the modulated enable signal while two input lines are dedicated to the rotation sense control. Other digital lines are available to detect board

⁷<http://www.pololu.com/catalog/product/707>

⁸VNH2SP30-E devices developed by ST Microelectronics

faults such as overcurrent and over temperature. Two analog outputs can be used for obtaining a measure of the instantaneous current absorption of each motor. The interface board is directly connected to the digital input and output lines of the microcontroller.

The digital outputs coming from the encoders are level shifted in order to meet the correct input voltages of the microcontroller by means of a custom made interface board. The stabilized 5 V supply for the motor board and for the encoder devices is obtained from the 12 V nominal supply provided by the lead acid batteries by means of a custom switching DC-DC voltage regulator board with a maximum current rating of 1 A. Another similar voltage regulator board is adopted to obtain the 6 V supply line needed by the microcontroller boards.

A custom power distribution board is installed on the robot principal frame in order to provide fuse protected 12 V outputs for motors and external devices. The board can connect up to four 12 V lead acid batteries managing their recharge. A tension clamping and a polarity inversion protection circuits protect all the electronics devices, and the power board itself, from overvoltages and batteries connection faults.

6.2.1 Microcontrollers

Two identical microcontrollers are installed on the BART robot: the STM32 F103 32-bit microcontrollers developed by ST Microelectronics. The microcontrollers are based on a Cortex M3 RISC CPU with a frequency of 72 MHz. 128 Kbytes of FLASH memory are available for storing persistent data and code and 20 Kbyte of RAM are packed on the same chip. The microcontrollers offer a wide range of integrated peripherals, in particular: up to 80 I/O ports (the available number depends on the mapping configuration), three 16 bit configurable timers with PWM output mode, 2 watchdog timers, a 24 bit system counter, a 16 bit PWM generator and two 12 bit A/D converters. Nine communication interfaces are available: two I²C interfaces, three universal synchronous/asynchronous receiver-transmitter (USART), two SPI interfaces, a CAN 2.0B interface and an USB interface. The greatest part

of the integrated peripherals can be interfaced with the DMA controller and can be associated to an advanced interrupt controller with the possibility of managing nested interrupts on the basis of their associated priority.

The slave microcontroller installed on the differential drive block is mounted on a motherboard (Olimex STM32-P103 board) that provides the necessary power supply circuitry, external oscillators, control leds, integrated circuits for interfacing the microcontroller with several communication physical layers such as the CANBUS, RS-232 serial and USB, and a small prototyping area. The entire board is power supplied by a 6 V DC line.

The master microcontroller is mounted on a custom made motherboard that is quite similar to the Olimex board already described. In addition it offer a larger number of connectors both for interfacing hardware and for debug purpose.

ST Microelectronics provides an open driver library which comprises the drivers for each integrated peripheral. Each integrated device can be activated, deactivated and configured as regards its functioning parameters by exploiting a set of functions and predefined parameters that mask the actual low-level register configuration to the developer. For what regards the I/O devices, besides configuring the peripheral, a proper configuration of the input and output lines is needed too. After the configuration, the firmware of the microcontroller can be developed by filling the interrupt handlers functions and by implementing the main function provided by the library. The peripheral driver library is implemented and distributed in pure ANSI C language.

The firmware sources are cross-compiled and transferred on the microcontroller FLASH memory by means of a third-party free and open source toolchain. To this purpose the Open On-Chip Debugger Toolchain has been used. This tool provides an Eclipse-based set of plugins for compiling, linking and debugging source code, and for loading the binary implementation of the firmware on the microcontroller FLASH memory.

The toolchain is composed of several independent programs:

Eclipse IDE The Eclipse platform⁹ offers an integrated environment for software development. It was originally designed for the development of Java programs although a large number of plug-ins has extended the possibility of using Eclipse for other programming languages.

Zylin CDT plug-in The Zylin-CDT¹⁰ plug-in is a third party version of the standard Eclipse CDT plug-in for C/C++ development. The Zylin-CDT is specifically developed to assist the developer in the the cross-compilation of software for microcontrollers. In this case firmware was developed on a windows 32 bit machine and then compiled for an ARM 32 bit machine.

GNU Tools The Zylin-CDT plug-in exploits the *GNU binutils*, the *GNU Compiler Collection* and the *GNU Debugger* for compiling, linking and debugging source code. In particular the YAGARTO¹¹ redistribution of the *binutils* and the *debugger* are used together with the *Sourcery G++ Lite* edition from *Codesourcery*¹² of the *GNU Compiler Collection*.

OpenOCD OpenOCD¹³ is a set of tools for the FLASH programming, the functional test and the on-chip debugging of microcontrollers. It provides the means for interfacing with the microcontroller through the JTAG (Joint Test Action Group) interface. To this purpose, a commercially available USB to JTAG adapter was adopted.

6.2.2 Embedded PC

Together with the microcontrollers, the robot is equipped with a single-board compact embedded PC (Compulab's Fit-PC2). The machine is based on an Intel Atom 1.6 GHz 32-bit CPU (x86) with 1 GByte of RAM memory. It can be power supplied by a 12 V unregulated source directly coming from robot's batteries. The entire PC is packaged into an aluminium package with

⁹<http://www.eclipse.org>

¹⁰<http://opensource.zylin.com/embeddedcdt.html>

¹¹<http://www.yagarto.de>

¹²<http://elinux.org/ARMCompilers>

¹³<http://openocd.sourceforge.net/>

very small dimensions (104 x 100 x 23 millimeters including connectors and button) and the weight is of 90 g. The power consumption is rated from 6 to 9 W depending on the load. The small dimensions, the low weight and the low power consumption made this PC suitable for being installed on the battery-powered mobile robot.

The FitPC2 offers six USB interfaces, a SD memory slot, an IrDA interface, an Ethernet interface, a 802.11 WiFi interface and HDMI video output. The hardware is compatible both with Microsoft Windows and Linux-based operating systems, for this robot, a 32 bit distribution of Ubuntu 12.04 operating system has been installed and configured.

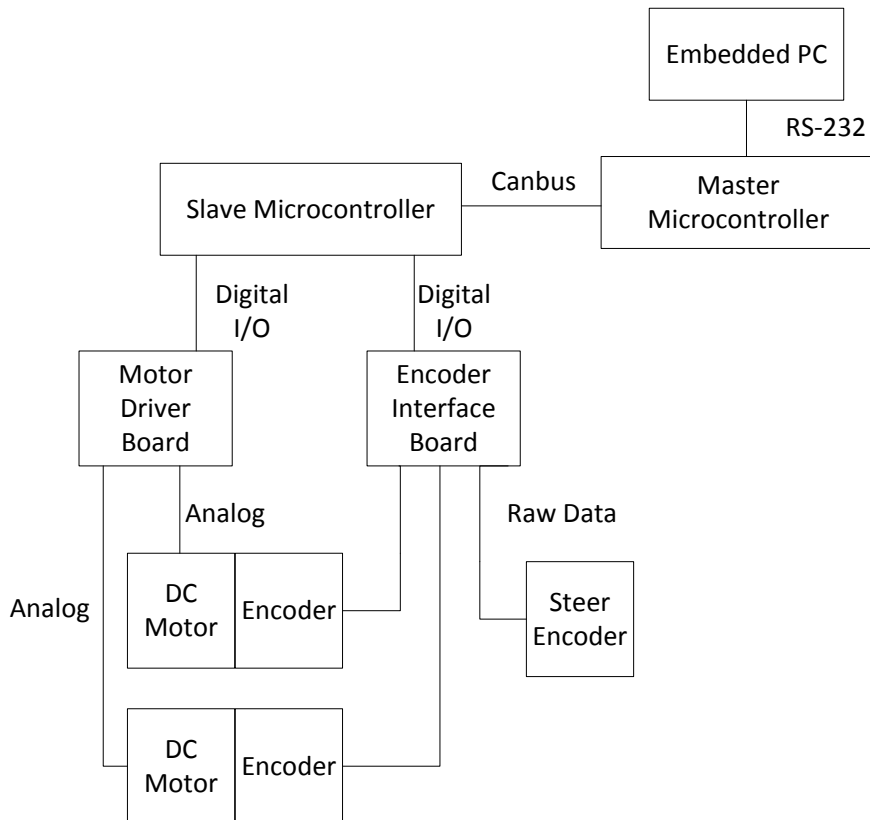


Figure 6.3: The overall electronic architecture of the BART robot.

The Figure 6.3 summarizes the overall electronic architecture of the BART robot.

The slave microcontroller is interfaced with the hardware devices through its digital I/O channels and interface boards, the communication among the slave and the master microcontroller is provided by the Canbus protocol which allows a robust and deterministic mechanism for data exchange and execution timing, the bitrate is 1 *Mbit/s*. The master microcontroller exchanges data with the embedded PC by means of a standard RS-232 serial line adopting a custom communication protocol and a bitrate of 115200 *bit/s*. The embedded PC can communicate with other remote machines through its 802.11 WLAN interface.

6.3 Control Software

The control software of the BART robot is distributed among four units of computation: the master and the slave microcontrollers, the embedded PC and the remote machine. This distribution of the functionality leads to two important issues: first, the communication among the units have an important and critical importance, especially considering the real-time requirements related to the control of the robot. Second, the heterogeneity of the involved computational units force to design software in a more flexible way.

In designing and developing the control software of the robot we exploited the integration between the Robot Operating System (ROS) and the Service Component Architecture (SCA) presented in Chapter 5. As already stated, this provides the possibility to exploit the strengths of both the approaches.

ROS offers the possibility to easily define components with a periodic behaviour and provides a set of tools that can strongly speed-up and simplify the development and the debugging of robotic applications. Moreover, it provides a wide set of libraries for interfacing different hardware devices, such as sensors and actuators, and for higher-level functionality such as pose tracking, navigation, mapping and so forth.

On the other hand, SCA provides an extremely flexible environment supporting the seamless distribution of software components among several networked computational nodes making it possible a scenario in which many nodes can cooperate for a common goal, each one of them offering a specific

service. Moreover, it cannot be ignored that the use of a SCA environment, such as Apache Tuscany, gives the possibility to fully exploit the portability of the Java language and environment.

The use of such a high-level programming language is not a common practice in robotics and this can be attributed to performances issues. Though Java performances can get closer to C++ performances in some cases [51], the Java environment still lacks in predictability. This lack is mainly caused by the unavoidable presence of the *garbage collector*. Although several attempts to make Java effective in embedded and real-time domain have been proposed, mainly involving the use of a real-time Java Virtual Machine and a real-time capable operating system, the Java language is not spread used in embedded domain nowadays. In such kind of domains, the developers prefer to adopt lower-level languages such as C++ and, often, ANSI C. This strongly influences the flexibility and, above all, the portability of software.

By using Java, instead, a greater level of portability can be reached and, additionally, one can exploit the wide set of libraries, framework and tools allowing to interface with databases, web-based resource, advanced networking and so forth.

In this work we tackle these issues by splitting the functionality on different nodes deployed into specialized environments, this allows us to take advantage of the strengths of each one of them while minimizing the drawbacks.

In particular, hard real-time computation is confined into the two micro-controllers that directly interface the hardware devices providing the low-level and time critical computations needed for the correct handling of control algorithms. Slightly higher-level computation, with less rigid time constraints, are carried out by the embedded PC through the use of the ROS framework. The highest-level functionality that do not need for real-time execution but, instead, can be computation intensive, are carried out by one or more remote machines.

In particular, these higher level activities are those that can take a greater advantage in exploiting the flexibility that the use of a general-purpose component model as SCA can offer.

Concluding, the overall approach can be synthesized in splitting the func-

tionality and adopting the software instruments that best fit their particular requirements.

We some details about the design and the implementation of individual computational nodes are given.

6.3.1 Slave Microcontroller Firmware

The main tasks of the slave microcontroller firmware are: controlling the speed of the actuated wheels on the basis of the speed references provided by the master and periodically give to the same master the functioning state of the differential drive block.

The speed control algorithm of the wheels is carried out by two independent PID controllers that are synchronously executed with a frequency of 500 Hz. The correct execution period is obtained by a properly configured internal timer of the STM32 microcontroller. The timer, in turn, obtain its operating frequency from the system frequency defined by a physical oscillator by means of a configurable frequency divisor. At defined and constant time intervals, the timer triggers its associated interrupt and the associated handler function is called as well. In this function the implementation of the two control algorithms is provided.

The speed feedback is obtained by properly decoding the changes of the logical states of the digital inputs ports connected to the outputs of the encoder interface board. Each microcontroller's digital input is associated, by properly configuring the peripheral driver, to an interrupt handler that is sensitive both on rising and falling edges of the signal. By correctly interpreting the state transitions it is possible to decode both the angle increment and the direction of the rotation. This is carried out by implementing a quadrature reading strategy for each encoder, as introduced in Section 6.1.

The firmware observes three encoders, two are associated to motors and hence, this feedback, is used to estimate the current speed of the wheels, providing the speed feedback to the closed-loop controllers. The third encoder measures the steer angle.

The control output, periodically generated by the controllers, is used to

generate a PWM output for the motor driver board. On the basis of the sign of the output, a couple of digital outputs for each motor are set for imposing the rotation sense or the braking modality to the motors. The PWM waves are actually generated by a properly configured timer offering this Specific functionality.

Finally, at the end of each control step, a new Canbus frame containing the current measured speeds and the steer position is generated and sent to the master microcontroller.

The slave firmware strongly exploits the Nester Vectorized Interrupt Controller (NVIC) peripheral of the microcontroller. Each interrupt line, and the affiliated interrupt handler, can be associated with a priority level. If during the execution of a handler, an interrupt with a higher priority occurs, the execution is interrupted in order to handle the new interrupt and returns when the handling has finished. On the other hand, if an interrupt with a lower priority occurs, the current handler is kept in execution and the new interrupt is stored in a queue. The interrupt queue is ordered on the basis of the priority.

This mechanism is made efficient by a particular functionality offered by the STM32 architecture that manages the change of registers context from one interrupt to another by means of a dedicated set of low-level assembly instructions. The context switch, that involves the storing of all CPU registers, is carried out in about 150 ns on a CPU running at 72 Mhz.

6.3.2 Master Microcontroller Firmware

The main tasks of the firmware running on the master microcontroller are the control of the steer position of the differential-drive block, the inverse kinematics computation and the pose estimation of the robot (odometry). Moreover, the master microcontroller implements a bridge between the slave and the embedded PC that plays the role of *supervisor*.

All master's activities are carried out by periodic functions since the master do not have to interact with physical devices. The overall timing is provided by the slave microcontrollers by means of the incoming Canbus

messages sequences, this provides a tight time-coupling between the two microcontrollers and avoids the need for a difficult and critical high-precision frequency synchronization between the devices.

The robot position estimation, known as odometry computation, is carried out at a frequency of 500 Hz. The computation is triggered by the arrival of a new Canbus message from the slave carrying new data representing the estimated wheel speeds and the steer position. On the basis of these data, and given the mechanical structure and the dimensions of the robot, the master recomputes the estimation of the robot pose. The pose is expressed by means of the rototranslation transformation between the robot relative frame, centered on the robot centre, and the fixed environment reference frame. At the system start, the two frames are overlapping.

Since the odometry estimation is based on the numerical integration of wheels speed, the uncertainty of the estimation is constantly growing. However, it is not the task of the microcontroller to correct this estimation since this operation involves the use of higher-level functionality such as localization and mapping. Typically, these higher-level task are carried out by taking into account other sources of measurements, for example, laser scanners, sonars, inertial platforms or vision-based pose estimation. The needed filtering and data fusion algorithms, such as the extended Kalman filter or the particle filter, are typically computation intensive and, in most systems, are carried out by high-powered CPUs mainly located on remote machines. In order to support the refinement of the odometry estimation, the estimated position made by the master microcontroller can be externally reset.

The motion control of the robot is a task that involves both control and kinematic inversion and it is executed periodically with a frequency of 100 Hz. According to the last robot speed reference (twist) received on the serial interface, the target wheel speeds and the goal steer angle are obtained by computing the inverse kinematics. However, these references cannot be directly set to the slave microcontroller since it must be guaranteed, at any time, that the kinematic constraint must be respected. As we discussed in Section 6.1.2, there must be an unique Instantaneous Centre of Rotation

(ICR). This point is defined both by the velocities of the wheels and by the current position of the steer, in any case, these two values must be coherent. On the contrary, if wheels speeds are not coherent with the current steer position, the constraint is no more respected and this can lead to wheels slipping, excessive motor effort or both.

For this reason, the controller is in charge of continuously monitor the steer position and the wheels speeds and estimate the ICR position in order to respect the kinematic constraint and, at the same time, follow the given reference. Figure 6.4 shows the overall controllers architecture involved in the robot control. The robot target is expressed as a couple of values V_{robot} and ω_{robot} . The two speeds provided as output by the *inverse kinematics* block (ω'_R and ω'_L) are used as inputs by the *robot controller* block that produces the actual references for the control loops (ω_R and ω_L). This operation is carried out by estimating the ICR position from the estimated wheel speeds ($\omega_{R,est}$ and $\omega_{L,est}$) and the steer position ϕ_{steer} . The closed loop PID controllers use the wheels speed feedback estimated from the position readings of the encoders (P_L and P_R).

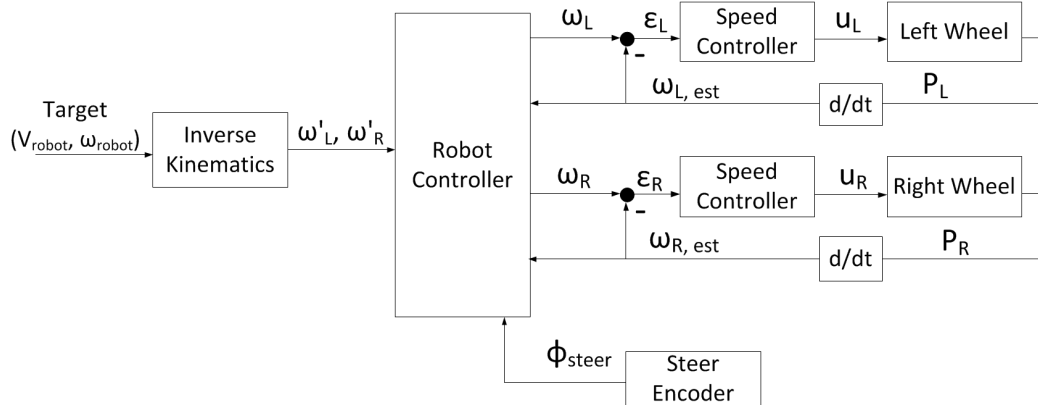


Figure 6.4: The controllers architecture.

The master microcontroller sends the new speed references to the slave with a frequency of 100 Hz through the Canbus, at the end of each control period. The communication from the master to the embedded PC is triggered every 100 ms (10 Hz). The data packet sent on the RS-232 communication interface

contains the odometry estimation and other data useful for monitoring the operational state of the robot. The data transmission takes a long time, due to the poor bandwidth offered by the serial interface. For this reason, the send operation is carried out by artificially triggering a timer interrupt handler with a low priority. The execution of this handler is interrupted by all other handlers and, hence, its execution runs in background filling the time intervals in which the microcontroller CPU would be in idle state.

In addition to this, the master microcontroller's firmware manages the initialization phase of the robot by properly sending target speeds to the actuated wheels in order to find the steer zero position. This operation is needed at the system power-on since the steer encoder is incremental, although provided with an additional signal indicating the zero position. Since, at the power-on phase, the steer position is not known, the master sends two reference speeds to the differential-drive block in order to make it rotate on place. When the encoder reaches the zero position, the steer is moved back to the centre and the robot enters in fully operational state.

Communication

As mentioned above, the communication between the slave and master microcontroller is carried out by Canbus interfaces. This interface was chosen due to the great robustness that it offers, this feature become more important considering that the wires carrying data have to pass through the slip ring collector that, inevitably, injects electronic noise. The outstanding determinism of the Canbus communication can be exploited by using the message exchange as a mean to synchronize the execution flow of the two microcontrollers.

However, the use of Canbus communication has some drawbacks, in particular when taking into account the payload dimension that is limited to only eight bytes. This obliged to define an algorithm for serializing data into small data packets that would fit into the payload of a single message. Figure 6.5 shows the structure of the two messages: the first represents the message sent from the master to the slave microcontroller, the second is the message sent from the slave to the master. Each data field has been serialized into a

ID	Right Wheel Target Speed (LSB)	Right Wheel Target Speed (MSB)	Left Wheel Target Speed (LSB)	Left Wheel Target Speed (MSB)	n/a	n/a	n/a
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7

ID	Right Wheel Speed (LSB)	Right Wheel Speed (MSB)	Left Wheel Speed (LSB)	Left Wheel Speed (MSB)	Steer Position (LSB)	Steer Position (MSB)	State
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7

Figure 6.5: The structure of the payload of Canbus messages.

couple of bytes allowing for a 16 bit resolution for data representation, that is sufficient for the purpose. Each message carries an identifier that uniquely identifies the type of message that can be useful for defining new messages for future works.

START	MSG ID	CHECK	PAYLOAD	PAYLOAD	...	PAYLOAD	END
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	...	Byte 14	Byte 15

START	MSG ID	CHECK	PAYLOAD	PAYLOAD	...	PAYLOAD	END
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	...	Byte 18	Byte 19

Figure 6.6: Serial communication messages.

The communication between the master microcontroller and the embedded PC is managed by a serial RS-232 interface. This interface is still widespread on modern PC and embedded computers and, if a serial interface is not available, it can be easily replaced by a USB to serial converter as in this

case. The use of the RS-232 can beneficially impact on the flexibility of the hardware configuration since the embedded PC can be substituted with another machine, for example a more powerful embedded computer, or another microcontroller or it can be removed at all by making the serial communication wireless by means of a commercial available serial to ZigBee or Bluetooth converter.

However, the use of RS-232 has in low bitrate its major drawback and, in this application, the same serialization algorithm adopted for the Canbus has been implemented. In this case, however, not for limiting the payload dimension needed for the data transfer but for limiting the overall amount of data to be transferred. In Figure 6.6 we show the serial message structure. The first message represents the packet sent from the embedded PC to the master microcontroller containing the velocity set-point, the second is the message that periodically the master sends to the supervisor containing the robot state. Each message contains a checksum field computed on the basis of the message payload that is checked by the receiver in order to detect the presence of corrupted data, and a message identifier code that identifies different possible type of messages. Each message has also a recipient field that can be useful in case of experiments with multi-master or multi-supervisor alternative scenarios. Messages starts and ends with two predefined codes that simplify the parsing operations when they are received by recipients.

6.3.3 Embedded PC Software

The supervisor software running on the embedded PC is implemented as a ROS node in the C++ programming language. Its main goal is to supervising the robot functioning and providing a bridge between the robot hardware and the higher-level tasks.

The software has a dedicated thread of execution for handling incoming messages from the serial port. The thread waits for the arrival of new bytes on the port and, once a complete message has been received, it proceeds with the verification of the message structure and the checksum field. If the message is correct, the thread updates the local representation of the robot

state stored in a dedicated class.

The use of a dedicated thread for the serial communication handling provides time decoupling between the ROS node execution and the data arrival timing allowing for an increased flexibility when defining the operational frequencies of the microcontroller firmware and the supervisor node. The thread is created and managed by using the POSIX Thread APIs that are part of the Portable Operating System Interface for Unix (POSIX) standard IEEE 1003.

The ROS supervisor node is fully asynchronous, when new data is received from the robot's microcontrollers, it generates the output messages that are written on the node's output topics. In particular, the node publishes two separate topics: the `bart_odometry_output` and the `bart_state_output`. The odometry topic exchanges odometry standard messages provided by the ROS navigation stack (`nav_msgs`), Listing 6.1 shows the declaration of the odometry message type in the ROS message definition language (MDL).

As many other ROS messages, the `Odometry` contains the nested definition of other messages, in this case the `PoseWithCovariance` and the `TwistWithCovariance`, their declaration is shown in Listings 6.2 and 6.3 respectively.

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Listing 6.1: The `nav_msgs/Odometry` message.

The `geometry_msgs/PoseWithCovariance` message represents a position in the free space with uncertainty, the position itself is stored in the `Pose` field, while the uncertainty is represented by the 6×6 covariance matrix stored in the array of floating point values.

Similarly, the `geometry_msgs/TwistWithCovariance` represents a twist, that is a representation of the velocity in the free space with uncertainty.

As for the `geometry_msgs/PoseWithCovariance` message, the uncertainty is expressed by means of a 6×6 covariance matrix.

The `Pose` message contains a ROS `Point` message and a ROS `Quaternion` message. The `Point` represents the position of a point in a free space by means of three parameters (x, y, z) , while the `Quaternion` message is the representation of the orientation in the free space in quaternion form (x, y, z, w) .

```
Pose pose
float64[36] covariance
```

Listing 6.2: The `geometry_msgs/PoseWithCovariance` message.

The `Twist` message contains two vectors of three elements each, the first, called `linear` represent the linear part of the velocity in free space, while the second, called `angular`, contains the angular part of the velocity.

The ROS supervisor node receives from the master microcontroller the robot state containing the odometry information which comprises the instantaneous speed estimation of the robot. Once the message is received and correctly parsed, it is converted into an `Odometry` message and published on the `bart_odometry_output` topic.

```
Twist twist
float64[36] covariance
```

Listing 6.3: The `geometry_msgs/TwistWithCovariance` message.

The incoming message from the microcontroller contains also the current operational state of the robot that can indicate failures, emergency or messages expressing other situations. In this case data is interpreted and published on the `bart_state_output` that exchanges ROS standard string messages.

The supervisor node is subscribed to two different input topics: the `bart_twist_input` and the `bart_command_input`. The first is used to send

to the supervisor new target twists while the second is available for sending other kind of messages, for example for triggering the reset of the odometry. The `bart_command_input` topic exchanges standard ROS strings while the `bart_twist_input` transports the already mentioned `Twist` message.

Given the kinematic structure of the robot presented in Section 6.1.2, the twist command allows two fields only: The linear velocity along the x axis of the robot and the rotational velocity around the z axis of the robot.

All presented messages contains a `Header` field that stores information regarding the timestamp, that represent the instant of time associated to the message according to the ROS global time, and an incremental sequence number.

When a new message is received on the input topics, the node reacts by calling a specific callback function that allows to retrieve the message content. The message is then interpreted, packaged into a serial message, and sent to the master microcontroller.

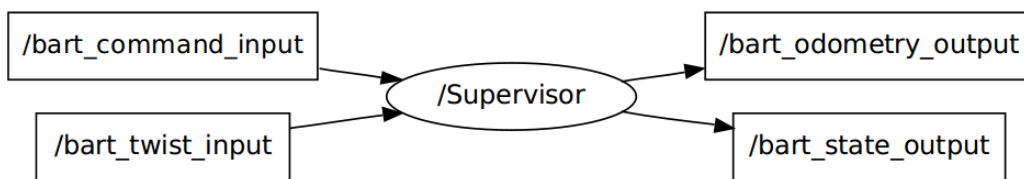


Figure 6.7: Graph node of the supervisor component.

The overall behaviour of the supervisor node is fully event-driven and all actions are triggered by the arrival of new messages, both from the master through the serial port, and from the other higher-level nodes on the input topics. In Figure 6.7 the portion of the ROS graph node representing the execution of the supervisor component is shown.

6.3.4 Remote Workstation Software

The software running on the remote workstation is implemented as a SCA composite. It contains two separate components: `RosGateSCA` and the `BartClient`. The `RosGateSCA` is the component that implements the ROS-

SCA integration as presented in Chapter 5. The `BartClient` is a fully fledged SCA component that, in this example, gives to the user a graphical interface for sending twist commands to the robot and receiving the odometry and state information, in addition, it offers a keyboard based console for the remote control of the robot.

In the example, the components are executed on a single remote machine although it is possible to deploy components on different machines in such a way that is fully transparent for the component implementation. For example, the `RosGateSCA` could be executed on the robot embedded PC or on a different machine and the `BartClient`, that offers a GUI, could be deployed on a portable device such as a tablet PC.

The machine runs an Apache Tuscany SCA domain that represent the actual instance of the SCA run-time environment, it is in charge of connecting components services to references, setting the properties and providing the connection for the exchange of data.

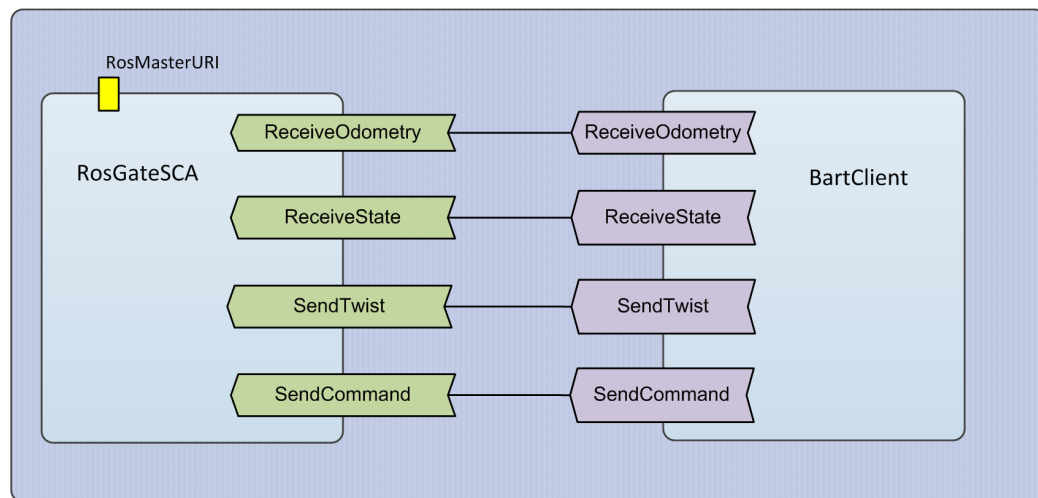


Figure 6.8: The composite diagram of the ROS-SCA integrated system.

The interactions between the two components are modeled and realized as reference-to-services interactions, the `RosGateSCA` component provides four services for interacting with ROS topics while the `BartClient` declares four corresponding references. The exchange of data is carried out by means of

transfer objects which encapsulate the content on ROS message in a ROS-unaware manner. By doing so, no knowledge, and hence no dependencies, regarding the ROS environment are needed outside the `RosGateSCA` component that, in turn, encapsulate all information related to the interaction with the ROS environment. Each transfer object is implemented as a storage class which provides methods for setting and retrieving its content without adding any kind of computation to the contained data.

Figure 6.8 shows a graphical representation of the composite diagram as defined in the XML-SCDL composite file reported in Listing 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:t="http://tuscany.apache.org/xmlns/sca/1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="BART_NS"
  autowire="false">
  <component name="RosGateSCA">

    <implementation.java
      class="rosGateSCA.impl.RosGateSCAImpl"/>

    <service name="ReceiveOdometry">
      <interface.java
        interface="rosGateSCA.intf.ReceiveOdometry"/>
    </service>

    <service name="ReceiveState">
      <interface.java
        interface="rosGateSCA.intf.ReceiveState"/>
    </service>

    <service name="SendTwist">
      <interface.java
        interface="rosGateSCA.intf.SendTwist"/>
    </service>

    <service name="SendCommand">
```

```
<interface.java
  interface="rosGateSCA.intf.SendCommand"/>
</service>

<property name="RosMasterURI" type="xsd:string">
  http://bart-FitPC2:11311
</property>

</component>

<component name="BartClient">

  <implementation.java
    class="bartClient.impl.BartClientImpl"/>

  <reference name="ReceiveOdometry"></reference>

  <reference name="ReceiveState"></reference>

  <reference name="SendTwist"></reference>

  <reference name="SendCommand"></reference>

</component>

<wire source="BartClient/ReceiveOdometry"
  target="RosGate/ReceiveOdometry"/>

<wire source="BartClient/ReceiveState"
  target="RosGate/ReceiveState"/>

<wire source="BartClient/SendTwist"
  target="RosGate/SendTwist"/>

<wire source="BartClient/SendCommand"
  target="RosGate/SendCommand"/>

</composite>
```

Listing 6.4: The remote workstation composite file.

Before analyzing the two components in detail, it is important to notice that the use of services to model the interaction with topics differs from the the interaction semantics provided by ROS topics. When a client of the `RosGateSCA` component wants to publish a message on a topic, it can simply invoke the operation provided by the corresponding service. On the other hand, if the client wants to read data from a topic, it has to invoke the corresponding method of the defined service. This changes the semantics of ROS topics since communication is no more asynchronous and the client is no more notified when a new message is available. It is a client's task to query the `RosGateSCA` component to obtain new messages.

This feature has two important consequences: first it strongly enforces the time-decoupling between the execution of the `RosGateSCA` component and the execution of its clients allowing for an improved flexibility, second, it requires the implementation of a mechanism for handling messages since more than one message can be received between two consecutive requests from the client. This mechanism can be an ordered queue, LIFO or FIFO, or any other kind of structure: the most appropriated strategy should be implemented by the developer according to the semantics of the messages and the application task. In this case, the `MessageManager` class is in charge of handling the messages received from the robot and, in this particular implementation, it keeps in memory only the most recent message for each topic.

The `RosGateSCA` component is implemented by the class `RosGateSCAImpl` contained in the implementation package. The component provides four services:

- **ReceiveOdometry**: this service provides means to retrieve the last odometry message sent from the robot. The implementation is defined by the Java class `rosGateSCA.intf.ReceiveOdometry`
- **ReceiveState**: this service allows clients to retrieve the last state message sent from the robot. The implementation is defined by the Java class `rosGateSCA.intf.ReceiveState`
- **SendTwist**: this service provide means to send a new twist message to the robot. The implementation is defined by the Java class

```
rosGateSCA.intf.SendTwist
```

- **SendCommand**: this message allows clients to send a command to the robot, for example for resetting the odometry estimation. The implementation is defined by the Java class `rosGateSCA.intf.SendCommand`

The only required property (`RosMasterURI`) is a string containing the URI of the machine which is currently running the ROS master.

The `BartClient` component is implemented by the Java class contained in the implementation package (`BartClientImpl`). It requires four references that, in this particular case, correspond to the four services of the previous component. When declaring the references, the specific interface that the required service must implement is explicitly indicated. For this reason, any other component that offers a service which implements the specified interface can be connected to the reference and vice-versa.

The “autowire” option is kept disabled, this makes the specification of the wiring between services and references mandatory. As it is shown in the last part of the XML-SCDL file, for each wire, both source component and reference names and the target component and service names are specified.

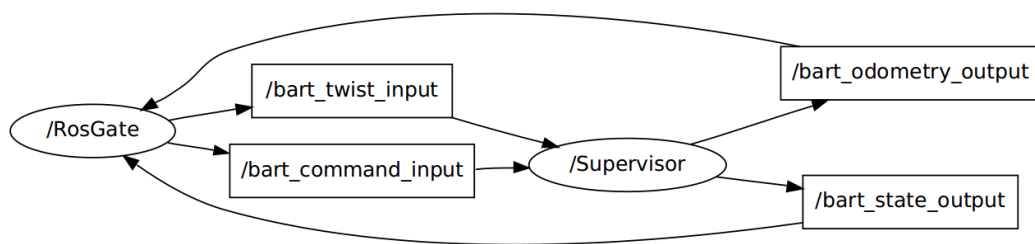


Figure 6.9: The ROS graph node of the ROS-SCA integrated system.

Figure 6.9 shows the ROS graph node representation of the integrated system from the ROS point of view. The `RosGate` and the supervisor node interacts by means of the four already mentioned topics.

The RosGateSCA Component

The RosGateSCA component embodies the link between the ROS subsystem and the SCA environment. Its main goal is to encapsulate all ROS related issues in a single component that offers an unique access point to ROS independent clients.

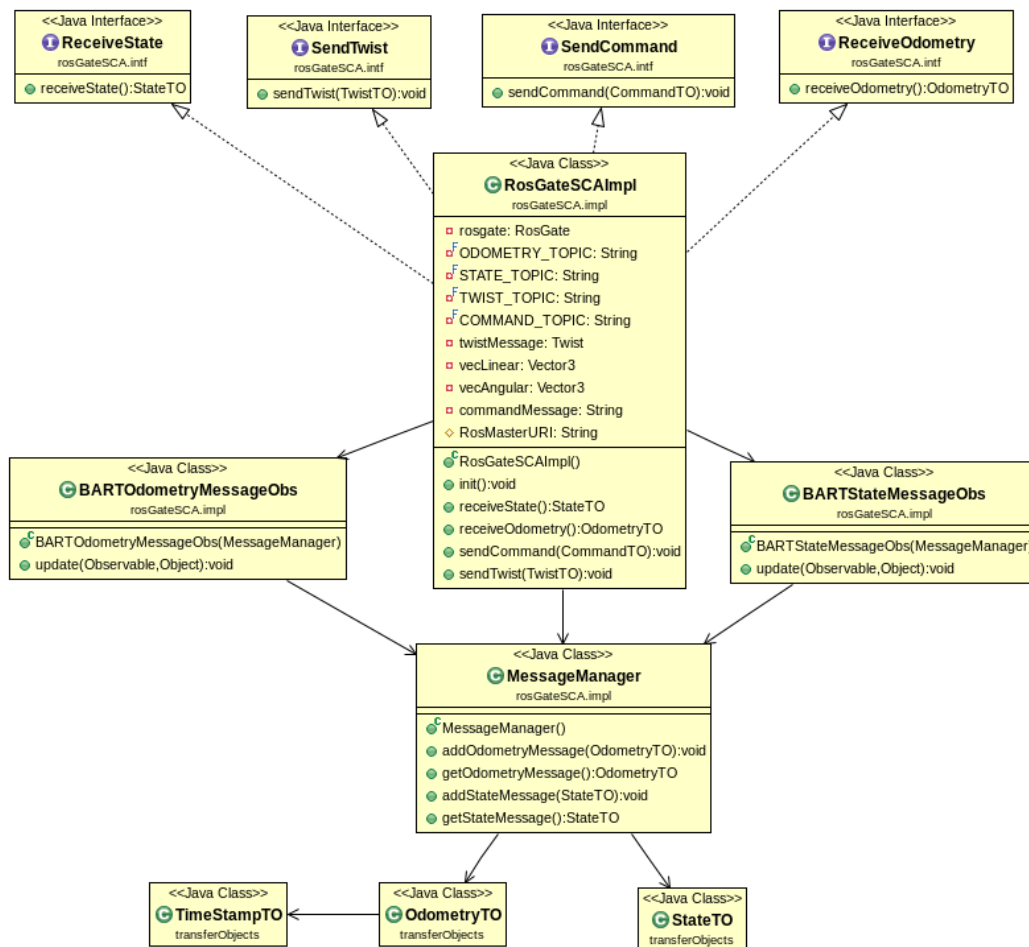


Figure 6.10: The RosGateSCA component class diagram.

Figure 6.10 shows the class diagram of the component. As defined in the composite file (Listing 6.4) the RosGateSCAImpl class represents the implementation of the RosGateSCA component. The class implements four interfaces that define the four services provided by the component.

The class is associated to two observers: the `BARTodometryMessageObs` and the `BARTStateMessageObs`. These two observers are in turn associated to the input topics subscribed by the component as described in Section 5.3.3. Each one of them implements the standard `Observer` Java interface and they are notified every time a new message is received on the respective topics.

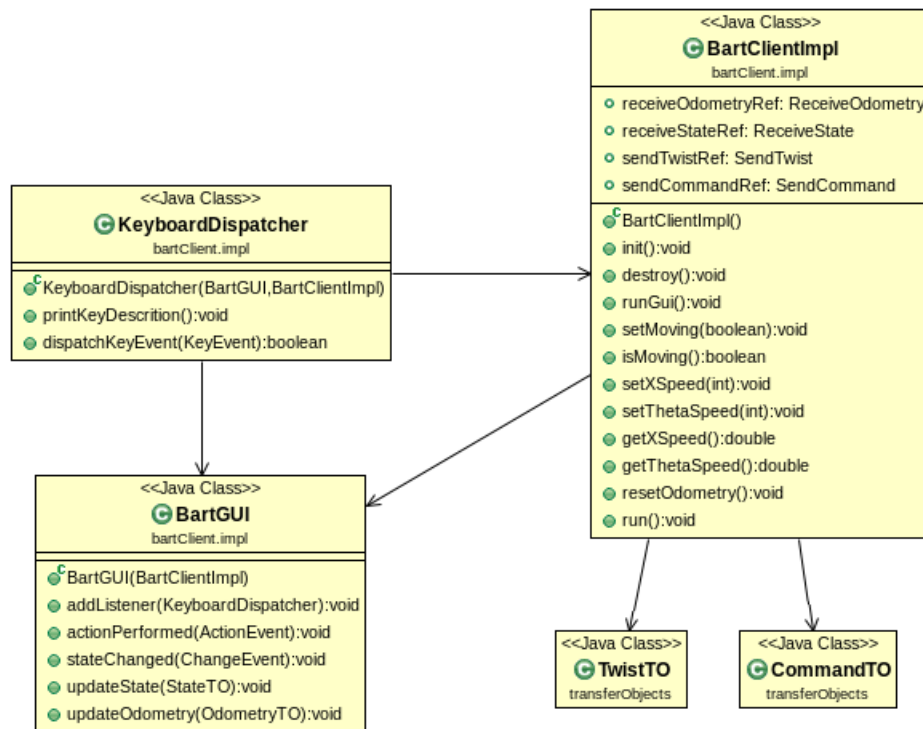


Figure 6.11: The `BartClient` component class diagram.

The `MessageManager` class provides means for managing the incoming messages and for encapsulate the content of the received ROS messages into the associated *transfer objects* implemented by the classes `OdometryTO` and `StateTO`. The `TimeStampTO` class is kept apart since this kind of object can be used by all ROS messages that carry time information. Fields and methods have been hidden from the diagram for the sake of space.

Listing 6.5 shows a portion of the `RosGateSCAImpl` class, the code has been simplified for the sake of space.

```
1 @EagerInit
2 public class RosGateSCAImpl implements SendTwist, SendCommand,
   ReceiveOdometry, ReceiveState{
3     private RosGate rosgate;
4     private MessageManager msgMan;
5     private BARTOdometryMessageObs odoObs;
6     private BARTStateMessageObs stateObs;
7     @Property(name="RosMasterURI", required=true)
8     protected String RosMasterURI;
9     private std_msgs.String commandMessage;
10    @Init
11    public void init(){
12        rosgate = new RosGate(RosMasterURI);
13        msgMan = new MessageManager();
14        odoObs = new BARTOdometryMessageObs(msgMan);
15        stateObs = new BARTStateMessageObs(msgMan);
16        rosgate.createListener("bart_odometry_output",
17            nav_msgs.Odometry._TYPE);
18        rosgate.addObserver("bart_odometry_output",this.odoObs);
19        rosgate.createListener("bart_state_output",std_msgs.String._TYPE);
20        rosgate.addObserver("bart_state_output",this.stateObs);
21        rosgate.createPublisher("bart_twist_input",geometry_msgs.Twist._TYPE);
22        rosgate.createPublisher("bart_command_input",std_msgs.String._TYPE);
23        rosgate.setup();
24        /* Message Creations... */
25        commandMessage = (std_msgs.String)
26            rosgate.createMessage(std_msgs.String._TYPE);
27    }
28    /* Services implementations... */
29    public StateTO receiveState() {
30        return msgMan.getStateMessage();
31    }
32    public void sendCommand(CommandTO command) {
33        commandMessage.setData(command.getCommand());
34        rosgate.publishMessage("bart_command_input", this.commandMessage);
35    }
```

Listing 6.5: The RosGateSCAImpl class.

The class implements the four interfaces that define its provided services: `SendTwist`, `SendCommand`, `ReceiveOdometry`, `ReceiveState`. The `@EagerInit` annotation at line 1 tells to the Tuscany SCA run-time to initialize the component as soon as possible, without waiting for the first invocation of a service. The SCA property at lines 7 and 8 represents the URI of the `roscore` node that it is used by the `RosGate` object to notify itself to the `roscore` at line 12. Lines 14 and 15 show the creation of two observer objects that will be associated to subscribed topics: `"bart_odometry_output"` and `"bart_state_output"`. From line 16 to 22 the listing shows the creation of *listeners* and *publishers* objects and the association of the aforementioned observers to the subscribed topics. The `setup` method called at line 23 initialize the `RosGateSCAImpl` object. Lines 25 and 26 show the creation of a `String` message by exploiting the message factory functionality of the `RosGate` class, the creation of other messages has been omitted.

From line 29 to 31 the implementation of the `receiveState` method of the `ReceiveState` service is shown. This service is used by clients to retrieve the state of the BART robot. It is implemented by requiring the corresponding data from the `MessageManager` which, in turn, receive data from the observer associated to the `"bart_state_output"` topic, as shown in line 15.

Lines from 32 to 35 show the implementation of the `sendCommand` method of the `SendCommand` service. The information is extracted from the transfer object received as parameter and store into the ROS Java message obtained from the `RosGate` class. Then, the message is published on the `"bart_command_input"` topic to be transmitted to the robot driver. Other services implementations are conceptually similar and have been omitted for the sake of space.

The `BartClient` Component

The `BartClient` component implements a GUI that offers to the user the possibility of sending twist commands to the robot by means of a graphical interface or by using a standard keyboard. The same interface shows the odometry data and robot state to the user. Figure 6.11 depicts the UML

class diagram of the component. The `BartClientImpl` class implements the component, the `BartGUI` class realizes the graphical user interface while the `KeyboardDispatcher` handles the keyboard events coming from the user.

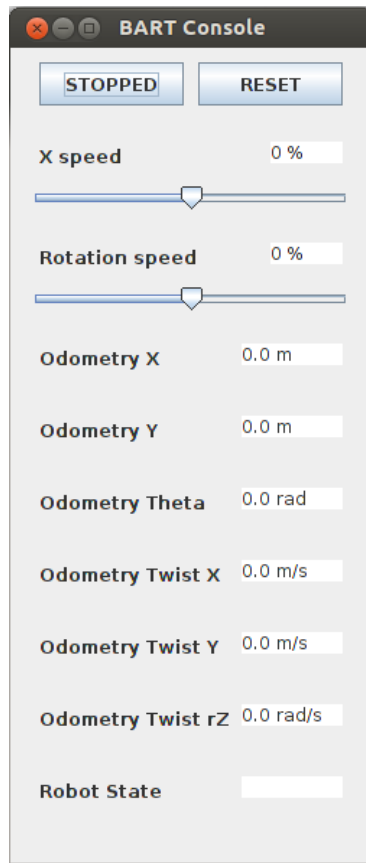


Figure 6.12: The GUI provided by the `BartClient` component.

The component acts as a client for the `RosGateSCA` component which provides the services for interacting with the robot. In particular, the `BartClient` execution is timed by a thread that periodically polls the `ReceiveState` and the `ReceiveOdometry` services in order to acquire new messages from the robot and update the console.

With the same frequency, the `BartClient` invokes the `SendCommand` and the `SendTwist` services for sending to the robot new commands and velocity references if the user has provided new inputs through the console depicted

in Figure 6.12.

It is important to notice that the client implementation is completely independent from any ROS related concept or issue. The exchange of data is carried out by means of the `TwistT0` and the `CommandT0` which encapsulate the ROS message content into ROS-independent classes.

Conclusions

Component Based Software Engineering (CBSE) allows the realization of modular and flexible software systems with a marked focus on software reusability. In the robotics field the CBSE principles struggle to become spread used. This is mainly due to the particular features of the robotic domain where the extreme variability in functionality, applications and involved hardware directly impacts on the component implementation and, by consequence, on their reuse capability.

We believe that the *component model* concern plays a fundamental role in defining the rules that supervise the components definition and the interaction among them. Components definitions and interactions strongly influence the flexibility of software systems and their capability to cope with the unavoidable variability of the robotic domain. Hence, the component model features directly influences the flexibility of the system.

The comparison between different available component models and their classification represents a fundamental step for understanding their features. The comparison can be carried out both with the objective of choosing the most appropriate component model with reference to the particular needs and with the aim of defining a new component model.

Whatever the objective, a good classification strategy should be general enough to be able to analyze and classify heterogeneous models catching their commonalities and specific enough to effectively capture their specific features.

We have proposed a classification and analysis technique that exploits the separation of concerns as we consider such an approach as the key to obtain a good classification framework. The classification is split into four orthogonal dimensions or concerns: the *Communication*, the *Computation*, the *Configuration* and the *Coordination*. Each one captures a particular aspect of the component model nature.

Each concern is then analyzed across a number of sub-dimensions that further specify and refine the analysis. By means of the feature model method, the containment relationships among features can be formalized in a clear and unambiguous manner keeping the classification simple and compact.

We analyzed five component models that represent five different and iconic approaches. By means of the developed classification we managed to clearly identify the differences and the commonalities among them making clearer the difference between domain-specific and general purpose component models. The analyzed component models are: the Service Component Architecture (SCA), the Corba Component Model (CorbaCM), the OpenCOM component model, the OROCOS component model and the Robot Operating System (ROS).

From the analysis and the classification of the five component models we have drawn the conclusion that all the desirable features that maximize the flexibility of a system cannot be achieved by adopting one single component model. We proposed the integration of component models as a method to overcome this limitation. In particular we integrated the ROS component model and the SCA component model by means of a bridge software component that connects a ROS-based subsystem with a SCA-based subsystem in such a way that this integration is transparent to each others.

We demonstrated the effectiveness of this approach by applying it to a new mobile robot, the BART. This robot provides a distributed computation architecture that enables the developers to experiment several different design approaches. In particular, the robot is provided with two microcontrollers, an embedded PC and at least one remote workstation composing an heterogeneous test system both for what regards the variability in computational architecture and for what regards the communication infrastructure.

When designing and realizing the control software of the BART robot we exploited the proposed integration strategy and we demonstrated that this integration, besides being possible, increases the flexibility of the system and gives to the developers the possibility of reusing preexisting component implementations or developing new ones. It also allows for the use of programming language that best fits the task or the computational environment such as ANSI C for microcontrollers, C++ for the supervisor and Java for higher-level tasks on remote machines.

7.1 Future Work

The work presented so far can be extended in several directions, both related to the classification of component models and for what regards the integration between ROS and SCA.

The classification can be further refined while maintaining the the “4C” separation of concerns that represent the core of the entire approach. Each one of the four concerns can be extended by adding new features or by specializing the already present ones. However, a “good” classification should be enough precise to fit well all the analyzed component models and, possibly, others that are yet to come. At the same time, the classification should remain as generic as possible since a complex classification scheme, with a very deep hierarchy of features and sub-features become difficult to use and it is unlikely to be able to clearly classify new component models. The trade off between generality and specificity is the main issue of all classifications and the proposed one does not represent an exception to this.

The classification activity can be further carried on by extending the analysis to other component models in literature. Up to now we analyzed models with the goal of finding weakness, strengths and differences between general-purpose component models and embedded domain specific models but this is only one of the many possible comparisons that can be made. We think that the proposed classification is complete and general enough to allow many other kinds of comparisons, for example between components that are just theoretical proposals or proofs of concept and industrially used component

models, such as Koala or Pecos to name a few, with the aim of finding the issues that prevent a large diffusion of some models in the industrial field

Also the integration technique can be improved and extended in several ways. A possible future direction can be the definition of a standardized mechanism for configuring the RosGate component that, up to now, still requires some manual coding operations for the definition and the setup of topics. A possible solution can be the definition of a XML-based language for the definition of the RosGate configuration that makes the setup of such a component automatic while preventing the users to go into the implementation details.

The Apache Tuscany run-time environment can be extended as discussed in Section 5.1. This feature could be effectively exploited for including the ROS-SCA integration directly in the run-time environment making this feature an integral part of the Tuscany environment.

Since Tuscany can support the execution of C++ components by means of a particular implementation of the run-time environment (called *native*), the possibility of directly integrating C++ native ROS components into C++ SCA components could be pursued. This could lead to a easier integration since the *rosjava* client library should not be needed anymore. As a side effect, the overall performances could be improved allowing the use of SCA components also for more time constrained tasks.

Finally, the use of SCA allows the exploitation of advanced tools and libraries that, in some cases, are well suited for the robotics domain. As an example, in [34], the use of Abstract States Machines (ASM) [32] is exploited for the definition of a software coordinator for robotic related task. The ASM formalism allows for formal and rigorous definition of tasks without the mathematical overkill typical of many other formal methods. To this purpose, in [61], an extension to the ASM graphical notation (flowchart) is proposed. This extension, called *pattern-oriented control-state ASM* allows the developers to define *patterns* of interactions that can be reused and specialized in many different applications. This considerably speeds-up the development process of an ASM specification when used to define coordination mechanisms for software components.

Bibliography

- [1] Apache Tuscany web page. <http://tuscany.apache.org/>, 2013. (Cited at pages 79 and 127)
- [2] Fabric³ web page. <http://tuscany.apache.org/>, 2013. (Cited at pages 79 and 132)
- [3] IBM Websphere web page. <http://www.ibm.com/software/websphere/>, 2013. (Cited at pages 79 and 126)
- [4] Java Orchestration Language Interpreter Engine (JOLIE) web page. <http://www.jolie-lang.org/>, 2013. (Cited at page 116)
- [5] Linux Works web page. <http://www.linuxworks.com/>, 2013. (Cited at page 35)
- [6] Object Management Group web page. <http://www.omg.org/>, 2013. (Cited at pages 27 and 52)
- [7] OpenCV library web page. <http://opencv.willowgarage.com/wiki/>, 2013. (Cited at page 143)
- [8] Organization for the Advancement of Structured Information Standards (OASIS) web page. <http://www.oasis-open.org>, 2013. (Cited at page 116)

-
- [9] OW2 - FraSCAti web page. <https://wiki.ow2.org/frascati/Wiki.jsp>, 2013. (Cited at pages 79 and 129)
- [10] QNX web page. <http://www.qnx.com/>, 2013. (Cited at page 35)
- [11] Real Time Application Interface (RTAI) web page. <https://www.rtai.org/>, 2013. (Cited at pages 35 and 96)
- [12] Rosjava documentation web page. http://docs.rosjava.googlecode.com/hg/rosjava_core/html/index.html, 2013. (Cited at page 115)
- [13] RT Linux web page. <https://rt.wiki.kernel.org>, 2013. (Cited at page 35)
- [14] Service Component Architecture (SCA) web page. <http://www.oasis-openca.org/sca>, 2013. (Cited at pages 76 and 115)
- [15] Service Oriented Architecture standard (SOA) web page. <http://www.opengroup.org/subjectareas/soa>, 2013. (Cited at page 116)
- [16] SOAP protocol web page. <http://www.w3.org/TR/soap12-part1/>, 2013. (Cited at page 116)
- [17] The Gazebo simulator web page. <http://gazebo.org/>, 2013. (Cited at page 143)
- [18] The Orocos Project web page. <http://www.orocos.org>, 2013. (Cited at pages 61 and 143)
- [19] The Player project web page. <http://playerstage.sourceforge.net/>, 2013. (Cited at page 143)
- [20] The Robot Operating System web page. <http://www.ros.org>, 2013. (Cited at pages 70 and 133)
- [21] Universal Description, Discovery and Integration standard (UDDI) web page. <http://uddi.xml.org/>, 2013. (Cited at page 116)

-
- [22] W³C Web Services Description Language (WSDL) web page. <http://www.w3.org/TR/wsdl>, 2013. (Cited at page 116)
- [23] Web Services Business Process Execution Language (WSBPEL) technical committee web page. <https://www.oasis-open.org/committees/wsbpel/>, 2013. (Cited at page 116)
- [24] Wind River VxWorks RTOS web page. <http://www.windriver.com/products/vxworks/>, 2013. (Cited at page 35)
- [25] Xenomai: real-time framework for Linux web page. <http://www.xenomai.org/>, 2013. (Cited at pages 35 and 96)
- [26] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The save approach to component-based development of vehicular systems. *J. Syst. Softw.*, 80(5):655–667, 2007. (Cited at page 74)
- [27] F. Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1122:1 – 18, 1998. (Cited at page 46)
- [28] C. Atkinson, J. Bayer, O. Laitenberger, and J. Zettel. Component-based software engineering: The kobra approach. In *ICSE 2000, 22th International Conference on Software Engineering, 3rd Workshop on Component-Based Software Engineering*, 2000. (Cited at page 58)
- [29] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: the kobra approach. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 289–309, Norwell, MA, USA, 2000. Kluwer Academic Publishers. (Cited at page 58)
- [30] D. Bálek and F. Plášil. Software connectors and their role in component deployment. In *3rd Int. Working Conf. New Developments in Distributed Applications and Interoperable Systems*, pages 69–84. Springer, 2002. (Cited at page 48)

- [31] Y. Bontemps, P. Heymans, P. Schobbens, and J. Trigaux. Semantics of foda feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, pages 48–58, 2004. (Cited at page 16)
- [32] E. Börger and R. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer, 2003. (Cited at page 192)
- [33] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck. Towards component-based robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005. (IROS2005)*, pages 163–168, August 2005. (Cited at page 64)
- [34] 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 at page 192)
- [35] D. Brugali and P. Scandurra. Component-Based Robotic Engineering (Part 1). *IEEE Robotics & Automation Magazine*, 9:84–96, December 2009. (Cited at page 28)
- [36] D. Brugali and A. Shakhimardanov. Component-Based Robotic Engineering (Part 2). *IEEE Robotics & Automation Magazine*, 10:100–112, March 2010. (Cited at pages 6, 9, 24 and 42)
- [37] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. (Cited at page 54)
- [38] H. Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001. (Cited at page 61)
- [39] H. Bruyninckx. Real-time and embedded guide. *KU Leuven, Mechanical Engineering*, 1:1–177, 2002. (Cited at page 47)

- [40] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the orocos project. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 2, pages 2766–2771. IEEE, 2003. (Cited at page 61)
- [41] T. Bureš, P. Hnětynka, and F. Plášil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proc. of SERA 2006*, pages 40–48, 2006. (Cited at page 81)
- [42] D. Calisi, A. Censi, L. Iocchi, and D. Nardi. Design choices for modular and flexible robotic software development: the openrdk viewpoint. *Journal of Software Engineering for Robotics*, 3(1):13–27, 2012. (Cited at page 14)
- [43] OMG Corba Component Model documentation web page. <http://www.omg.org/spec/CCM/>, 2013. (Cited at page 52)
- [44] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *In Proc. IASTED Software Engineering and Applications (SEA '04)*, 2004. (Cited at page 59)
- [45] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008. (Cited at pages 59, 101 and 103)
- [46] I. Crnkovic, M. Chaudron, S. Sentilles, and A. Vulgarakis. A classification framework for component models, 2007. (Cited at page 13)
- [47] K. Czarnecki. *Generative Programming: Methods, Techniques, and Applications Tutorial Abstract*. PhD thesis, Univeristy of Twente, The Netherlands, 2002. (Cited at page 16)
- [48] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003. (Cited at page 24)

- [49] The Fractal Project web page. <http://fractal.ow2.org/>, 2013. (Cited at page 54)
- [50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995. (Cited at pages 43 and 44)
- [51] L. Gherardi, D. Brugali, and D. Comotti. A java vs. c++ performance evaluation: a 3d modeling benchmark. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172, 2012. (Cited at page 167)
- [52] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K. Lundbäck. The rubus component model for resource constrained real-time systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008. (Cited at page 72)
- [53] G. Heineman and W. Councill. *Component-based software engineering: putting the pieces together*, volume 17. Addison-Wesley USA, 2001. (Cited at pages 1 and 2)
- [54] S. Hissam, J. Ivers, D. Plakosh, and K. Wallnau. Pin component technology (v1.0) and its C interface. Technical note, CMU/SEI-2005-TN-001, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, April 2005. (Cited at page 67)
- [55] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990. (Cited at page 15)
- [56] G. Kotonya, I. Sommerville, and S. Hall. Towards a classification model for component-based software engineering research. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 43–52, 2003. (Cited at page 12)
- [57] J. Kramer. Configuration programming - a framework for the development of distributable systems. In *CompEuro'90. Proceedings of the 1990*

- IEEE International Conference on Computer Systems and Software Engineering*, pages 374–384. IEEE, 1990. (Cited at page 38)
- [58] K. Lau, L. Ling, V. Ukis, and P. Velasco Elizondo. Composite connectors for composing software components. In *Software Composition*, pages 266–280. Springer, 2007. (Cited at page 48)
- [59] K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007. (Cited at page 10)
- [60] S. Laws, M. Combellack, R. Feng, H. Mahbod, and S. Nash. *Tuscany SCA in action*. Manning, 2011. (Cited at pages 79 and 127)
- [61] A. Luzzana, M. Rossetti, P. Righettini, and P. Scandurra. Modeling synchronization/communication patterns in vision-based robot control applications using asms. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 331–335. Springer, 2012. (Cited at page 192)
- [62] A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168, 2006. (Cited at page 64)
- [63] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000. (Cited at pages 28, 32 and 94)
- [64] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM. (Cited at page 48)
- [65] G. Moreno. Creating custom containers with generative techniques. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 29–38. ACM, 2006. (Cited at page 67)

- [66] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009. (Cited at pages 70 and 133)
- [67] J. Radatz. IEEE standard glossary of software engineering terminology. *IEEE Std 610121990*, 121990, 1990. (Cited at page 1)
- [68] M. Radestock and S. Eisenbach. Coordination in evolving systems. In S. LNCS, editor, *Proc. Int. Workshop Trends in Distributed Systems CORBA and Beyond*, volume 1161, pages 162–176, 1996. (Cited at pages 9, 21, 23, 30, 37 and 45)
- [69] C. Schlegel. *Navigation and Execution for Mobile Robots in Dynamic Environments - An Integrated Approach*. PhD thesis, University of Ulm, 2004. (Cited at page 79)
- [70] C. Schlegel. *Software Engineering for Experimental Robotics*, volume 30, chapter Communication Patterns as Key Towards Component Interoperability, pages 183–210. Springer, STAR series, 2007. (Cited at page 79)
- [71] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007. (Cited at pages 9 and 15)
- [72] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J. Stefani. Reconfigurable sca applications with the frascati platform. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 268–275. IEEE, 2009. (Cited at pages 79 and 129)
- [73] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 0:1–26, 2011. (Cited at pages 79 and 129)

- [74] SmartSoft web page. <http://smart-robotics.sourceforge.net>, 2013. (Cited at page 79)
- [75] SOFA 2.0 Project web page. <http://sofa.ow2.org>, 2013. (Cited at page 81)
- [76] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley, 2002. (Cited at pages 1 and 2)
- [77] R. Van Ommering, F. Van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000. (Cited at page 56)
- [78] N. Wang, D. Schmidt, M. Kircher, and K. Parameswaran. Towards a reflective middleware framework for qos-enabled corba component model applications. *IEEE Distributed Systems Online*, 2(5), 2001. (Cited at page 52)
- [79] N. Wang, D. Schmidt, and C. O’Ryan. Overview of the corba component model. In *Component-Based Software Engineering: putting the pieces together*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001. (Cited at page 52)
- [80] M. Winter, T. Genßler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, P. Müller, C. Stich, and B. Schönhage. Components for embedded software: the PECOS approach. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES02)*, pages 19–26, 2002. (Cited at page 69)