



**DOCTORAL SCHOOL  
ON LIFE AND HUMANOID TECHNOLOGIES**

# **Enhancing Software Module Reusability and Development in Robotic Applications**

**Ali Paikan**

Dissertation presented in partial  
fulfillment of the requirements for  
the degree of Doctor of Philosophy

February 2014

---

# Enhancing Software Module Reusability and Development in Robotic Applications

---

*Author:*

ALI PAIKAN

Dissertation presented in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy

Doctoral School on  
LIFE AND HUMANOID TECHNOLOGIES

*Supervisors:*

Dott. Lorenzo NATALE

Prof. Giorgio METTA

UNIVERSITÀ DEGLI STUDI DI GENOVA  
ISTITUTO ITALIANO DI TECNOLOGIA

GENOA , 2011–2014



## **Acknowledgements**

First and foremost, I would like to thank the promoters of this research work Lorenzo Natale and Giorgio Metta for giving me the chance to start my scientific career in the department of iCub Facility and Robotics Brain and Cognitive Sciences. It has been an excellent opportunity and indeed, an unforgettable experience. I cannot fully express my gratitude to my friend and supervisor Lorenzo, for truly understanding, guiding, encouraging and supporting me from the initial to the end of this project. We have gone through difficult moments, in which his patience and support have always kept me motivated and creative. The constructive advice, support and friendship of Giorgio, has been invaluable on both an academic and a personal level, for which I am extremely grateful.

This thesis would not have been possible without the help and encouragements of my colleagues from the department of iCub Facility. I would like to thank them for creating genuinely friendly environment in the lab and for their valuable technical comments and feedbacks.

I cannot fully express my feeling and special thanks to my family for their considerable support and for understanding me during all these years of my study. I am heartily and deeply thankful to my girlfriend Adda, simply for everything! She has taught me many things. She has always been there to cheer me up when things went sadly bad. In short, she has adequately defined the true meaning of sacrifice.

## **Abstract**

Software engineering and best practices promote modularity and composability to reduce debugging and development time of software applications in robotics. These approaches may increase the complexity of the system and the effort required to properly orchestrate the interactions between modules especially in distributed architectures. Coordination of components across different computers can easily lead to brittle systems and scalability problems unless an appropriate strategy is adopted. The contribution of this thesis has been divided in the three parts. The first part addresses the coordination problem of modules in distributed architecture and proposes an approach in which coordinating logic is transparently inserted into separated reusable components along with application-dependent data transformations. We demonstrate that by following our approach, coordination and arbitration can be carried out directly by exploiting connections that deliver data messages between modules. For this reason, it intrinsically reduces the number of links required for coordination and it can be built without changing existing modules.

The second part investigates how the extra requirement specific to an application can be added to an existing module as an extensible functionality. Systematically developing high-quality reusable software components requires careful design to find a proper balance between potential reuse, functionalities and ease of implementation. Extendibility is an important property for software which helps to reduce cost of development and significantly boosts its reusability. We introduce an approach to enhance components reusability by extending their functionalities using plug-ins at the level of the connection points (ports). Application dependent functionalities can be implemented using a conventional scripting language and

plugged into the ports of components. The main advantage of the proposed approach is that it avoids introducing application dependent modifications to existing components, thus reducing development time and fostering the development of simpler and therefore more reusable components.

The last part deals with the composition of modules in an application, their deployment and the implemented tools to support the application building and execution. Composition and deployment of distributed modules are time consuming and usually add a sensible overhead to the development cycle when manual activities are required. Some modules may need checking availability of specific hardware, computational resources or software libraries. That adds complexity to the execution of component on heterogeneous clusters of computers. Overall, this pushes the development of monolithic systems that are difficult to reuse and it prevents research in scenarios that require integrating behaviors involving cooperative activities of several sub-modules. We propose an extensible formalism of software components and applications in robotic. In the appendix, we finally describe some tools that rely on this formalism to support application development and management in the YARP framework.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Research Objectives . . . . .	3
1.2	Outline . . . . .	5
<b>2</b>	<b>Background and Positioning</b>	<b>6</b>
2.1	Finite State Machines . . . . .	6
2.2	Petri Nets . . . . .	8
2.3	Behavior-Based Systems (BBSs) . . . . .	10
2.4	Agent Programming Languages . . . . .	11
2.5	Coordination Models and Languages . . . . .	11
<b>3</b>	<b>A port-arbitrated mechanism for behavior selection in humanoid robotic</b>	<b>13</b>
3.1	Introduction . . . . .	13
3.2	Related work . . . . .	15
3.3	Arbitrating behaviors . . . . .	17
3.3.1	Computation of the activation values . . . . .	20
3.3.2	Representation of the rules . . . . .	21
3.3.3	Selection Mechanism . . . . .	22
3.4	Parameters tuning and design policy . . . . .	22

---

3.5	Experimental validation . . . . .	23
3.5.1	First experiment: Take an object . . . . .	23
3.5.2	Second experiment: Return an object . . . . .	25
3.5.3	Third experiment: Take and return . . . . .	25
3.6	Conclusions . . . . .	30
<b>4</b>	<b>Modeling robotic behaviors using port arbitration in YARP</b>	<b>31</b>
4.1	Introduction . . . . .	31
4.2	Related Work . . . . .	32
4.3	Representing arbitration rules using named connections . . . . .	33
4.4	Modeling the behaviors . . . . .	36
4.4.1	Behavior Specification . . . . .	37
4.5	Extracting Rules from Behavioral Model . . . . .	39
4.6	Reference Implementation in YARP . . . . .	39
4.7	Experimental Validation . . . . .	41
4.8	Conclusions . . . . .	44
<b>5</b>	<b>Extending Data Flow Port with Monitoring and Arbitration using Scripting Languages</b>	<b>46</b>
5.1	Introduction . . . . .	46
5.1.1	Motivating Example . . . . .	47
5.1.2	Contribution and outline . . . . .	49
5.2	Port Monitor Object . . . . .	50
5.2.1	Port monitor life cycle and API . . . . .	51
5.3	Port Arbitrator Object . . . . .	54
5.3.1	Internal Architecture of Port Arbitrator . . . . .	56



---

5.3.2	Representation and Evaluation of Constraints . . . . .	58
5.3.3	Reference Implementation . . . . .	59
5.4	Potential Applications . . . . .	60
5.4.1	Monitoring communication for QOS . . . . .	60
5.4.2	Data Guarding and Filtering . . . . .	62
5.4.3	Data Transformation . . . . .	63
5.4.4	Logging and Performance Monitoring . . . . .	63
5.5	Conclusions . . . . .	64
<b>6</b>	<b>Enhancing software module reusability using port plug-ins: an experiment with the iCub robot</b>	<b>65</b>
6.1	Introduction . . . . .	65
6.2	A step-by-step example . . . . .	67
6.2.1	Handling reachable objects . . . . .	69
6.2.2	Handling objects using tool . . . . .	72
6.2.3	Handling objects with human assistance . . . . .	74
6.3	Conclusions . . . . .	77
<b>7</b>	<b>Application description and management model in YARP</b>	<b>78</b>
7.1	Introduction . . . . .	78
7.2	Motivation . . . . .	80
7.2.1	Dependencies . . . . .	80
7.2.2	Interconnection . . . . .	81
7.2.3	Integration and composition . . . . .	81
7.2.4	Execution and monitoring . . . . .	81
7.2.5	Application migration and load balancing . . . . .	82

---

7.2.6	Cross–middleware deployment . . . . .	82
7.3	Conceptual Representation . . . . .	83
7.3.1	Resource . . . . .	83
7.3.2	Module . . . . .	83
7.3.3	Application . . . . .	85
7.4	System Architecture . . . . .	86
7.4.1	Dependency graph . . . . .	87
7.4.2	Dependency resolving and load balancing . . . . .	89
7.5	An example in YARP . . . . .	91
7.6	Conclusions . . . . .	93
<b>8</b>	<b>Conclusions</b>	<b>95</b>
8.1	Contribution . . . . .	95
8.1.1	Coordination of components in distributed architectures . . .	95
8.1.2	Extensibility and reusability of components . . . . .	96
8.1.3	Composition and deployment of components . . . . .	96
8.1.4	Software and Tools . . . . .	97
8.2	Discussion and Future works . . . . .	97
<b>A</b>	<b>Tools</b>	<b>100</b>
A.1	The gyarpmanager . . . . .	100
A.2	The gyarpbuilder . . . . .	102

# List of Figures

1.1	Chapters ordered according to the research objectives investigated in this work. . . . .	4
3.1	Two behaviors (The Face Detector and Object Detector) are connected to the input port of Gaze Control using connections $C_1$ and $C_2$ . The port arbitrator in Gaze Control coordinates the two behaviors using a set of rules and properties of each connection (dotted box). Please refer to the text for the definition of each symbol. . . . .	18
3.2	Computation of the stimulation and activation levels for each connection. $s[k]$ is the stimulation level at time $t_k$ . $\sigma$ is the stimulation gain and $\tau$ is the damping time. Stimulation is accumulated every time a new message arrives and it continuously decays over time. The connection is in active state once the cumulative stimulation reaches a threshold and until it gets completely discharged. . . . .	20
3.3	Configuration of the behavior network for “Take an object” and “Return an object”. . . . .	24
3.4	Configuration of behaviors for “Take and Return”. The connections that were previously used in “Take an object” and “Return an object” are shown here in gray. . . . .	26

3.5	iCub performing the “Take and return” experiment, <b>A</b> : looking for the object, <b>B</b> : reaching for the object, <b>C</b> : grasping the object, <b>D</b> : looking for a person while holding the object, <b>E</b> : approaching the person and <b>F</b> : releasing the object. . . . .	28
3.6	Arbitration in Arm Control during the “Take and return” experiment. The plots show the values of $S_i[k]$ and the selected connection by the arbitrator of all the $m$ input connections from the modules Release Detect, Grasped, Grasp Detect, Object Detector, Face Detector and Rest Arm (see Figure 3.4). Horizontal axis is time $t$ and vertical axis represents stimulation values $S_i[k]$ . Stimulation is plotted in green when the corresponding connection is selected and in orange otherwise. The bottom graph shows the action of the resulting behavior, i.e., <b>A</b> : looking for an object, <b>B</b> : reaching for the object, <b>C</b> : grasping, <b>D</b> : looking for a person, <b>E</b> : approaching a person and <b>F</b> : releasing the object. . . .	29
4.1	An example of different YARP components and the connections among them. The shaded box represents how an arbitrator is employed in the input port of a component to arbitrate between multiple connections to the same port. . . . .	34
4.2	An example of the behavioral model that uses the components from Figure 4.1 to implement the Search-and-Track behavior. This behavior allows the robot to look around in search for a face or an object. When the robot detects a face it tracks it with the gaze. When it detects an objects it follows it with the gaze and reaches for it. The overall behavior is implemented by coordinating simpler behaviors. Correct coordination is implemented by inhibitions among behaviors (red arrows). See also the description in the text. . . . .	37
4.3	Application generation from behavioral description. . . . .	40
4.4	The description of the behavior “Take and Return” implemented on the iCub robot. . . . .	43
5.1	Data-flow architecture of an object tracking application . . . . .	48

5.2	Different ways to provide required events for coordinator. . . . .	51
5.3	The life cycle of port monitor object. . . . .	52
5.4	Different ways to select desired data from multiple sources. . . . .	55
5.5	The architecture of Port Arbitrator object. Straight lines show the data flow and zigzag lines represent event flow. . . . .	57
6.1	The experimental setup and the simplified activity diagram of table-cleaning application. The reachable zone is depicted in green, the orange zone represents the zone reachable with the tool and finally the red zone indicates the unreachable space, for which the robot needs human intervention. . . . .	68
6.2	The iCub performing table-cleaning on reachable objects. The robot takes the object (A) and places it into to bucket (B). . . . .	70
6.3	Configuration of the modules for handling reachable objects on the table. . . . .	70
6.4	The iCub performing table-cleaning using a tool (rake). The robot take the tool (A), reaches for the object (B,C), pulls the object (D), grasps the object (E) and finally places it into the bucket (F). . . . .	72
6.5	Configuration of the modules for handling objects within tool-reach space. . . . .	73
6.6	The iCub performing table-cleaning with human assistance. The robot detects an unreachable object (A), detects the presence of a human and asks assistance (B,C), grasp the object (D) and finally places it into the bucket (E). . . . .	75
6.7	Configuration of the modules for table-cleaning application. . . . .	76
7.1	Conceptual representation of resource. . . . .	84
7.2	Conceptual representation of module and application. . . . .	85
7.3	Multilayer architectural design of Yarpmanager. . . . .	86

---

7.4	An example of graph used for dependency resolving. $A_i$ indicates an application. $M_i$ is a module, $R_i$ and $C_i$ are respectively required resource and computer, $o_i$ and $i_i$ are respectively output and input of a module. . . . .	88
A.1	A screenshot of gyarpmanager. . . . .	101
A.2	A screenshot of gyarpbuilder . . . . .	103

# Listings

4.1	The representation of "Be Curious" behavior in XML format. . . . .	40
5.1	Port monitor callback functions in Lua . . . . .	53
5.2	An example of monitoring data and dispatching events. . . . .	54
5.3	Port monitor extended API in Lua for arbitration . . . . .	57
5.4	Setting selection constraint and monitoring data from Template Matching. . . . .	60
5.5	An example of monitoring communication for QOS. . . . .	61
6.1	Monitoring and arbitrating connection $C_1$ . . . . .	71
6.2	Monitoring and arbitrating connection $C_2$ . . . . .	71
6.3	Monitoring connection $C_3$ for generating events. . . . .	71
6.4	Monitoring and arbitrating connection $C_4$ . . . . .	73
6.5	Monitoring connection $C_7$ and $C_{11}$ . . . . .	75
6.6	Monitoring and arbitrating connection $C_{10}$ . . . . .	76
7.1	Resource description in YARP. . . . .	92
7.2	Module description in YARP. . . . .	92
7.3	Application description in YARP. . . . .	93

# Chapter 1

## Introduction

Robotic community is continuing to grow. Within the community, researchers are developing increasingly complex humanoid robots which are aimed to be employed in unstructured and dynamical environments. Recent advancements in robotic research fields including mechanics, electronics, control, perception and machine learning are pushing back the frontiers toward achieving roboticists dream: a robot operates as human does in a human-like environment. However progress in individual domains is not enough to achieve this. An essential component which is usually overlooked is the software, which is what implements and integrates the individual components that make the overall robotic system.

A good software framework is indeed a key principle for developing a robust robotic system. Recent approaches to robot programming [15, 17, 54] push the idea that software should be organized in different components providing a well-defined, possibly, simple functionality, and that complex systems can be built by proper integration of a subset of these modules. Using Component-Based Software Engineering (CBSE) [38, 78], an application designer can compose software systems from a mixture of reusable off-the-shelf components. That significantly reduces the effort of developing new software applications by promoting the systematic reuse of existing solutions.

In the field of software architecture, there are five aspects of a component which should be carefully considered during the development of a reusable software. Those are



known as the rule of *Five Cs* or separation of concerns [4]: Computation (the computed functionality), Communication (how the computed result being communicated), Configuration (the parameters which defines the behavior of the component), Coordination (how component can be orchestrated) and Composition (how the components can be properly composed in the system). In a component-based system, these five concerns must be separated out to achieve a maintainable and reusable software.

Systematically developing high-quality reusable software components is a difficult task and requires a careful design. An important and challenging aspect of developing reusable software is the amount of functionality which should be offered by the component. A component can be implemented with the limited functionalities while meeting all of the software architecture standards. However, simplicity does not necessarily lead to more reusable software. On the other hand, with reusability in mind, there is a risk of *over-generalization* and increased complexity: to build a more generic and reusable component, the developer tries to foresee all the possible future needs and add them as reconfigurable functionalities to the software. Such a commitment may lead to more complex components, polluted with application-dependent functionalities that are more costly and difficult to maintain and use correctly. Thus, a proper balance must be found between potential reuse and ease of implementation [74].

The glue that holds the software together, the middleware, has a big impact on its viability. Ideally, middleware helps developing reusable software, since it factors out many details from components, leaving them simpler, cleaner, and more configurable. Many middlewares partially support the separation of the concerns (e.g., communications) with the prospective of being simpler, more flexible and light-weight to be easily used by inexperienced developers. These kind of frameworks usually do not constrain users to follow any specific component model, which may leave the components incompatible with the work of others. On the contrary, middlewares which rely on more complex component models usually require more knowledge, learning time and code development effort by non-experienced users. Another important issue concerning the development of reusable software is the specific architecture dependency. Experienced coders move as much functionality of a component as possible into a general-purpose library, keeping the middleware-using part light so it can be easier adapted with its future changes. Less experienced developer enthusiastically

use the middleware for everything and make their software tightly coupled to the middleware and specific programming paradigm. A well-behaved middleware should limit the downsides of that enthusiasm by providing a proper paradigm and tools for the systematic software development and maintenance.

This research study investigates the application development requirements in robotics and proposes some approaches to enhance reusability and the development of the software components in distributed robotic frameworks. The work mostly concentrates on the coordination of distributed modules, extensibility and reusability, composition and deployment of the components to provide the users with proper methods and tools for systematic application development for robot.

## 1.1 Research Objectives

The research objectives investigated in this work fall into three categories. The first part is related to addressing the problem of coordination of modules in distributed frameworks (e.g., YARP). The second part investigates how the extra requirements specific to an application can be added to an existing module as extensible functionality to foster the development of simpler and more reusable components. Finally, the last part deals with the composition of modules in an application, their deployment and the implemented tools to support the application building and execution. An overview showing these categories and the corresponding chapters is given in Figure 1.1. The rest of the section introduces the objectives of this work in terms of research questions which have been explored.

Software engineering and best practices promote modularity and composability to reduce debugging and development time of software applications in robotics. This approach, however, increases the complexity of the system and the effort required to properly orchestrate the interactions between modules. Coordination of distributed components across different computers can easily lead to brittle systems and scalability problems unless an appropriate strategy is adopted. Thus, the major area of focus is the coordination of complex robotic systems. More specifically, the following research questions is investigated in Chapter 3 and 4:

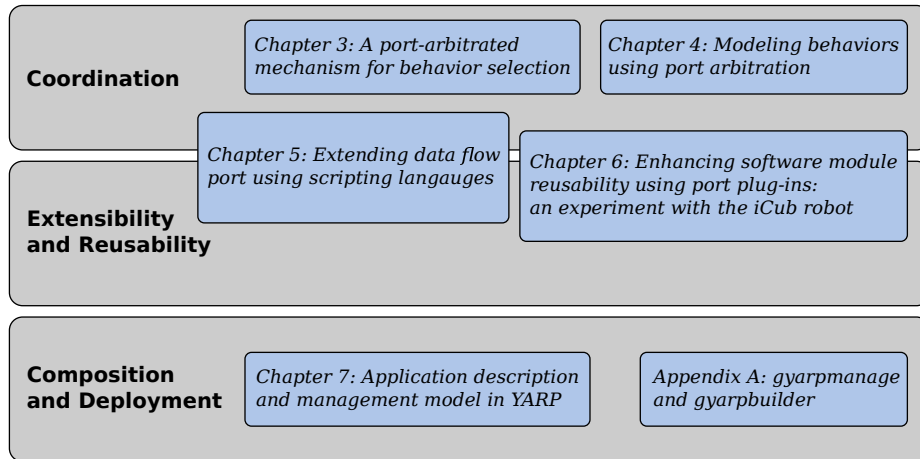


Figure 1.1: Chapters ordered according to the research objectives investigated in this work.

***Which coordination mechanism fits better the ongoing research objectives in the field of humanoid robotics and how it can be adopted by distributed robotic frameworks?***

A challenging part of developing a reusable software is the amount of functionality which should be implemented in a component. Usually, many application-specific functionalities are mixed with the computational services which causes to have more complex and error-prone components. The following research questions is investigated in Chapter 5 and 6:

***How application-specific functionalities can be separately added to an existing component to promote the development of simpler and more reusable software?***

Eventually components should be composed together in an application and run on a machine or remote machines. Composition and deployment of large number of modules are time consuming and they usually add a sensible overhead to the development cycle when manual activities are required. From a scientific point of view, this slows down the development of new ideas and prevents the study of complex scenarios that require integrated behaviors involving cooperative activities of several sub-modules. The following research questions is investigated in Chapter 7:

*How components can be modeled by an expendable formalism in order to be used for progressively development of tools which facilitate application design and deployment?*

## 1.2 Outline

This dissertation consists of 7 chapters and one appendix. Most of the core chapters are based on the peer reviewed papers submitted to scientific conference or journal.

**Chapter 2** explores the literature relevant to robot programming techniques and coordination mechanisms which is the major objective of this thesis.

**Chapter 3** presents a coordination mechanism based on port arbitration. Through describing a real robotic example and comparison with the existing approaches, the advantages of our work are demonstrated using YARP framework.

**Chapter 4** introduces a representation of robotics tasks using our port arbitration approach. The model is used to describe the behavior of applications in a more comprehensive manner and to facilitate the coordination of components.

**Chapter 5** concentrates on the extensibility of the components via port plug-ins and separation of computational components from application-dependent functionalities.

**Chapter 6** further demonstrates the potential advantages of port plug-ins and our coordination mechanism through presenting a step-by-step experiment with the iCub humanoid robot which is completely built out of the existing modules without code changes.

**Chapter 7** presents a formalism for modeling applications and component management in distributed architectures. The model is used for the development of composition (the "gyarpbuilder") and the deployment (the "gyarpmanager") tools for YARP middleware. A short description of the tools and their features is presented in Appendix A.

# Chapter 2

## Background and Positioning

The problem of controlling and coordinating the functional components of an autonomous robot's software to achieve a complex goal is a highly challenging task. Investigating a proper coordination mechanism for humanoid robotic application development and studying the functional requirements of components for the realization of that mechanism is one of the main focuses of this research work. Various control architectures and programming paradigms have been proposed in different fields (e.g. software engineering, artificial intelligence and computational neuroscience) which due to their special convenience for particular robot operational context and research objectives have been adopted only to a subset of robotics. The aim of this chapter is to provide a general overview of existing approaches for the programming of an autonomous robot control system.

### 2.1 Finite State Machines

Finite State Machine (FSM) is a mathematical model of computation which has been widely used to model a large number of problems in computer programs, communication protocol, artificial intelligence and many other fields. In a simple form, FSM uses states and transitions, and the states typically contain a set of actions which occur on the entry or exit of a state. An extension of FSM is hierarchical state machine (HSM) which has its origins in the state charts introduced by Harrel [36]. HSM allows that

states to be nested in a hierarchical manner meaning that it can be in multiple states simultaneously as long as those states have a parent-child relationship. Each state explicitly associated to one or more tasks and transitions represent the order that tasks can get activated and access the resources (e.g., robot actuators) to define the general behavior of the robot.

rFSM [45] or restricted Finite State Machine is a lightweight and a minimal subset of UML2 and Harel statecharts which consists only three model elements: states, transitions and connectors. It is designed to model coordination of robotic tasks and systems with a minimal number of semantic primitives. Instead of providing a rich set of built-in features for all possible use-cases, the rFSM model advocates dealing with complexity by composition. That means both local hierarchical composition of Statecharts or distributed composition as well as composition of core execution semantics with run-time extensions such as event memory. rFSM provides an extensible, framework independent and real-time safe implementation using Lua scripting language.

XRobots [80] is a domain-specific language for programming mobile robots based on augmented hierarchical state machines. In the language, states are treated as first class objects and thus they can be passed as arguments to other parametrized states. It also offers the template states which allow them to be customized and instantiated.

Smach [12], is a recent and widely used state machine implementation in ROS [71]. Smach is a library for task-level execution and coordination based on hierarchical concurrent state machines which aims to rapidly compose complex robot behaviors out of primitive ones. It introduces different container of states (e.g. StateMachine, Concurrent, Sequence and Iterator) thus has broader aim than being only a state machine. In contrast to the rFSM, Smach offers the concept of concurrent container which allows to define states with multiple child states which running simultaneously and the transition to another state can be done based on one or all of the child's outcome. Therefore, Smach differs from other previous approaches due to its support for having concurrent and distributed states.

## 2.2 Petri Nets

Generally, models based on finite state machines are inherently focused on the state of a system and the observable input–output behavior. They are not well suited to studying the interaction of concurrently active parts of a system and the combined behavior of distributed parallel systems [43]. This is an important fact since concurrency is source of complexity that can be rapidly overwhelming. Petri Nets [58], on the other hand, seems to be promising tools for modeling and analyzing the system which are characterized as being concurrent, asynchronous and distributed. A petri net usually consist of a finite set of places (possibly with some token inside them), transitions and arcs which link places to transition and vice versa. A transition is enabled if all input places connected to it contain a token and all output places are empty. These properties along with their strong mathematical representation has made petri nets popular for modeling robotic applications especially in the filed of industrial robotics.

Robotic Task Model (RTM) [62] is a framework for implementing robotic task co-ordination and evaluation from qualitative and quantitative viewpoints based on petri net. A RTM consists of a set of resources (e.g. a primitive task), a set of primitive action which robot can take to accomplish a task and a set of events which occurs upon finishing a task (i.e. generated by an action) or detecting an error condition. In RTM, resources are mapped to petri net places and transitions are associated to logical conditions defined over the events. A place which possesses a token corresponds to an available resource. When the resource is a primitive task, the token means that the corresponding primitive action (i.e. the action chosen to translate the primitive task) is running. Any logical condition associated to a transition is made true or false by the occurrence of events. Each primitive task may be actually implemented by more than one primitive action. This implies that, depending on the design, different primitive actions might be running concurrently and the transition occurs either one of them is performed.

The execution of coordinated tasks requires a mechanism for synchronization of actions and deterministic switch between sequential or parallel actions of the robot (i.e. motion coordination). [9] proposes a framework for the coordinated execution of tasks in humanoid robot based on petri net. The set of places describes the states of the

system (e.g. robot's arm is ready) and the set of transitions defines events (e.g. object is grasped) that can change these states. Occurrence of events or the execution of an operation may change the distribution of the tokens on places and put the system in a new state. The proposed approach also decomposes a task in different subsystems which allows them to run concurrently. Transition to another state can also take place under the condition that different subsystems are ready at the same time. Thus, according to [9], Petri nets can be used as an efficient tool for the coordinated motion control in robotic systems with a high degree of freedom such as humanoid robots.

The actual task implementation (i.e., its design and coordinated execution) requires the scheduling of the primitive tasks as well as the synchronization with the events. Implementing sequence of actions while handling all exceptions (e.g. an event due to a failure) results in immensely complicated network as the complexity of the task increases. [47] propose a petri-net model for task supervision in the field of humanoid robots. The proposed approach decompose the system in different petri nets: a network for execution of the sequence of actions and another one to deal with occurrence of exceptional circumstances (administration). The advantages is that the administration network does not need detail information about the actual sequence of actions. In another words, the administration has always identical network structure independent from the associated execution network.

RoboGraph [48] is a robot task programming IDE based on Signal Interpreted Petri Net (SIPN) editor [29] to program and coordinate the activity of modules written using CARMEN [57] middleware. The IDE supports programming in different levels. The first is to program tasks that must be executed autonomously by one robot and the second is to program tasks that can include several robots and building elements. The framework also consists of a centralized dispatcher to execute the Petri nets and a monitor that shows the state of all the running nets.

Petri-net marked languages are a superset of regular languages (based on FSA) which leads to a larger modeling power. One of the interesting point of modeling systems using petri nets is that they do not lead to the same state explosion as combining finite state machines [21]. Thus, It has been widely used for plan representation, analysis and decision making in large scale [90], [55], [22].



## 2.3 Behavior-Based Systems (BBSs)

The behavior-based paradigm tends to view an autonomous robot control system as a set of asynchronous and concurrent individual behaviors. The overall behavior of a robot is emerged as the result of the interactions of these behaviors among themselves (i.e. inhibition or collaboration) and with the environment. In this approach, there is no explicit representation of a robot's context, state of execution and action.

The original concept of BBS is Brooks' subsumption architecture [14] in which reactive behaviors are used in a multilayer system and behaviors from higher level (higher priority) can inhibit and suppress those in lower layers. However, this fixed-priority approach tends to run into scalability problems as the number of behaviors increases. Since then, various BBSs have been proposed which mainly differ in their approaches of coordinating the asynchronous behaviors of a robot. The coordination can be performed in a centralized or a distributed fashion using different mechanisms known as behavior selection methods [75]. The following presents iB2C [70] and DBN [44] as two representatives of BBSs. A comprehensive survey of behavior-based task coordination is given in [68].

Integrated Behavior-Based Control (iB2C) [70] offers an architecture of behavior-based systems which supports a wide variety of action selection and coordination mechanism such as priority-based and state-based arbitration, winner-take-all, superposition and voting. Coordination in iB2C is done by using separate signals for coordination (i.e. activation, stimulation, inhibition, target rating). Kertesz [44] introduces dynamic behavior network (DBN) based on stimulation, inhibition and excitation of the behaviors in a network. Behaviors in DBN use a set of preconditions and the stimuli from other connected behaviors to determine their actual states at any given time (normal, failed, activated, finished). DBN decomposes the problem in different subsystems which consists of static and dynamic behaviors. The key concept of DBN is that the behaviors can (dynamically) create new behaviors in their finishing or failing actions. Although concurrency is supported by DBN, the parallelism is simulated and the computation of the stimulus are done in iterations.

## 2.4 Agent Programming Languages

As a matter of choice, a robot control system can continuously interrogates the sensors and produce corresponding events whenever any change in the environment is perceived. An event handling mechanism can be employed to query the large number of events and produce required inputs for a task planner. In order to properly react to these changes toward achieving the main goal, the planer selects a proper subset of tasks to be executed. This leads us to another paradigm of robot programing known as Agent Programming Languages which is widely used in the filed of classical artificial intelligence.

One of the most suitable architectures for implementing deliberative behavior is the BDI architecture [64] inspired by the BDI (Belief-desire-intention) model of human practical reasoning [33]. This architecture includes components such as beliefs, goals, plans, and plan generating rules. Each plan generating rule specifies a plan reaching a goal if executed in a specific belief state. The deliberative behavior in BDI architecture is a cyclic process in which sensory information is processed, beliefs and goals are updated, applicable plan generating rules are selected, and applied, and then generated plans are executed.

Various agent programming languages have been designed and developed to facilitate the implementation of BDI architecture. Examples of these programming languages include 2APL [24], AgentSpeak(L) [72], Jason [13], and MetateM [10]. However, the application domains of these languages have been mainly limited to cognitive software agents. One reason for this might be due to the fact that the current agent programming languages lack necessary supports for addressing different requirements of robotic control systems. This urges system developers to invent many ad hoc solutions for such requirements, making development of such systems costly and hard to maintain.

## 2.5 Coordination Models and Languages

Early concurrent languages supported the interactions through shared variables which later have been extended to adopt message passing mechanism in distributed software

architecture. The coordination models and languages are originated from the work of Gelernter and Carriero [32] in the context of parallel and distributed systems and has had a large development in the last two decades. A coordination model provides a framework in which the interaction of active and independent entities can be expressed using a specific language. “A coordination model is the glue that binds separate activities into an ensemble” [32]. According to [6] a coordination language can be categorized based on two characteristics: data-orient or control-orient and endogenous or exogenous. Data-oriented coordination exploits the data flow and their interaction while a control-oriented coordination concerned with the activation and deactivation of control flow.

An example of endogenous coordination languages is Linda [31]. This language defines a mechanism to coordinate concurrent computations by means of messages which are formatted in tuple structure and can be added to the computation environment. They remain as named independent entities until some process chooses to receive them. Using some primitive offered by the language, the tuples can be read or written in a blocking or non-blocking manner, and new processes can be created to evaluate the tuples. Linda processes are decoupled from other processes by communicating only through the tuple space. However, endogenous languages have the fundamental drawback of intermixing computation with coordination [6].

In contrast to endogenous language which require computations to make use of specific primitives for coordination, exogenous language make use of coordination-agnostic computation. One of the most prominent example of this type of languages is Reo [7]. It is a paradigm for composition of distributed software components and services based on the notion of connectors. Reo enforces an exogenous channel-based coordination model that defines how designers can build connectors, out of simpler ones. Application designers can use Reo for compositional construction of connectors that coordinate the cooperative behavior of components in a component-based system. Thus, the coordination is a result of the topology of channels within the connector. A comprehensive and detailed survey of coordination languages is also given in [6].

## Chapter 3

# A port–arbitrated mechanism for behavior selection in humanoid robotic\*

### 3.1 Introduction

Recent approaches to robot programming in the literature [15, 17, 54] push the idea that software should be organized in modules each performing a well-defined, possibly simple, job and that complex tasks should be then solved by proper integration of a subset of these modules running concurrently. Integration and coordination in large systems is a challenging task. The typical approach is to delegate the coordination to special objects that manage the activities of the individual behaviors. Central coordinators are difficult to reuse and implement robustly. This solution easily leads to brittle systems and scalability problems.

Consider, as an example, the following behavior: a robot programmed to grasp an object and give it back to a person. A possible way to achieve this behavior is to decompose it into simpler behaviors (e.g. “Look for object”, “Reach for object”, “Grasp

---

\*This chapter is based on Ali Paikan, Giorgio Metta and Lorenzo Natale. A port–arbitrated mechanism for behavior selection in humanoid robotics. *The 16th International Conference on Advanced Robotics*. Nov 2013.

the object”, “Look for a person”, “Approach the person” and “Release the object”) and orchestrate them using a Finite State Machine (FSM) to obtain a sequence of actions. The FSM can consist of different states each activating one or more behaviors; the FSM change state when a specific event is triggered. For example, transition from state “Look for object” to “Reach for object” happens when event *Object found* occurs. From here event *Object is reached* brings the FSM to state “Grasp object”. Even in this simple example it may be difficult to design the FSM so that all conditions are properly handled. For example “Reach for object” and “Grasp object” should be active only if the object is visible. Returning the object should be done only if the robot has the object in the hand. This requires to continuously monitor that the object is in the hand using the available sensors and reset the FSM otherwise. Generally speaking each state in the FSM implicitly encodes the state of the external world (e.g. assuming that the object is in the hand while returning it to the person); a robust behavior requires that this state is either static or continuously monitored using the available feedback.

Programming robust applications considering reactivity, scalability and re-usability has always been at the center of attention of researchers. Different control architectures, such as deliberative, reactive or hybrid [60] have been studied in a wide variety of robotic domains. Among them, the behavior-based approach inspired from Brooks’ subsumption architecture [14] is of particular interest due to its fast response to external events. Traditionally, it has been used in robotic applications in which reactivity is crucial (e.g. [8, 86], see also section 3.2). However, it requires the addition of special connections that carry coordination signals; in distributed architectures these signals must also be synchronized with the ones that carry data.

We introduce a novel mechanism for coordination of behaviors in distributed architectures. In our approach modules are coordinated by arbitrating their connections. In practice, inhibition of a module is achieved by suppressing the data it receives. Because coordination relies on the same links which are already in place to transfer data among behaviors, this solution intrinsically reduces the number of required connections. In addition, since modules can receive data from multiple sources, it allows a finer degree of granularity (i.e., a module can inhibit only a subset of the connections of another module, thus allowing the latter to process data from other connections).

Finally, our approach can be implemented directly at the level of the middleware and it clearly separates code responsible for computation and coordination.

We implemented our approach using the YARP middleware [54] and tested it by developing a complex behavior on the iCub humanoid robot [53]. We show that using our approach we could implement a behavior that involves a sequence of actions by integrating pre-existing blocks and without the need to develop a special purpose module responsible for coordination. More importantly we developed our behavior incrementally.

The remainder of this chapter is structured as follows. The following section highlights some of the features of our proposed mechanism and compares it with other approaches. Section 3.3 presents the problem addressed in this chapter and describe the arbitration mechanism and its properties. Section 3.4 suggests some policies for tuning the connection parameters. In Section 3.5 we describe how we have used our approach for implementing a specific behavior on the iCub humanoid robot and in Section 8 we present our conclusion.

## 3.2 Related work

The problem addressed here has some similarities with the typical action selection problem studied in the field of ethology, neurobiology, computational neuroscience, artificial intelligence and robotics. The original concept of behavior-based system is Brooks' subsumption architecture [14] in which reactive behaviors are used in a multilayer system and behaviors from higher (priority) levels can inhibit and suppress others. However, coordinating behavior solely based on inhibition tends to limit the flexibility and reusability of the system [60]. To overcome this limitation, Maes [50] proposes a bottom-up selection mechanism in a non-hierarchical network of behaviors. Coordination in [50] uses three kinds of links between the behaviors (predecessor, successor and conflictor) and it works by adjusting the preconditions in which behaviors can operate. According to Tyrrell [81] and Hayashi et al. [37], this mechanism is not well suited for human-like action selection problems because the binary values used as precondition result in a loss of information. Hayashi et al. [37] also proposes an action

selection method based on motivation levels that resembles the dopamine system in animals. A continuous waveform of motivation signals are divided in different consciousness levels with some preassigned behaviors in each level. Based on the signal level, corresponding behaviors are chosen for execution. Since behaviors are statically prioritized, this approach imposes the same limitation of the subsumption architecture (i.e., a behavior may need to be in different consciousness level depending on the order in which it appears in a sequences of action). In contrast, our approach does not limit behaviors to be statically prioritized.

Different behavior selection mechanisms are compared in [49, 68] and [75]. An alternative approach to competitive action selection is a cooperative mechanism in which recommendations from multiple behaviors are combined to form a control action that represents their consensus. An example of this type of mechanism is DAMN [73]. It uses a centralized arbitrator to fuse the collected commands from different behaviors and select the action which best satisfies the prioritized goals of the system. Nowadays, due to heterogeneity of data types and the complexity of the control systems, the proposed methodology is practically limited to low-level control. The centralized coordination mechanism has been successfully used in different applications, but it can encounter scalability problems due to the overhead associated to transferring a relevant amount of information over several links to the coordinator [69]. In contrast our approach does not use any central coordinator. Inspired by voluntary action selection in humans [88], arbitration is done using regulated stimulation levels of the outputs of behaviors. Ayllu [84] is an architecture for distributed multi-robot behavioral control which allows standard port-arbitrated, behaviors interaction (message passing, inhibition, and suppression) to take place over IP networks. This architecture shares some concepts with our approach but it does not support important features such as stimulation and excitation.

Integrated Behavior-Based Control (iB2C) [70] is an architecture of behavior-based systems which supports a wide variety of action selection and coordination mechanisms such as priority-based and state-based arbitration, winner-take-all, superposition and voting. Coordination in iB2C is done by using separate signals for coordination (i.e., activation, stimulation, inhibition, target rating). That introduces extra links between behaviors and causes extra overhead that could be non-negligible in a

distributed system. In our approach, coordination is performed using properties of connections and it exploits links already used to transfer data thus intrinsically minimizing the overhead. Another advantage of our approach is that it can be implemented at the level of the middleware and it does not induce dependencies in the implementation of the individual modules.

Kertesz [44] introduces the dynamic behavior network (DBN) which has some similarities to our approach such as stimulation, inhibition and excitation of the behaviors in a network. Behaviors in DBN use a set of preconditions and stimuli from other connected behaviors to determine their actual states, at any given time (normal, failed, activated, finished). In contrast, our approach does not change any conditions or internal states of behaviors to activate or deactivate them. Instead, a behavior can decide whether to accept or not incoming data, based on the configuration of connections. Furthermore, our coordination mechanism allows behaviors to be really parallelized and distributed, in contrast to DBN where, parallelism is simulated and the computation of the stimuli is done in iterations. Similar to [70] and [44], in our approach, the complexity of the problem can be decomposed in different subsystems. In section 3.5, we show how, using our approach, a complex system is divided in subsystems that are configured, individually tested and finally combined together to implement the desired behavior.

### 3.3 Arbitrating behaviors

There is no concise definition of behavior in the literature. We refer to behavior as a computational unit with a set of preconditions and goals. It has a set of input ports to receive information from other behaviors and a set of output ports to stream out the result of its activity. Upon receiving data a behavior checks its activation conditions, performs an iteration step and sends the results through its output ports. We make the following assumptions for each behavior:

- The preconditions in which a behavior gets activated are local to the behavior itself and they are not visible to other behaviors. In other words, behaviors cannot directly activate or deactivate others.



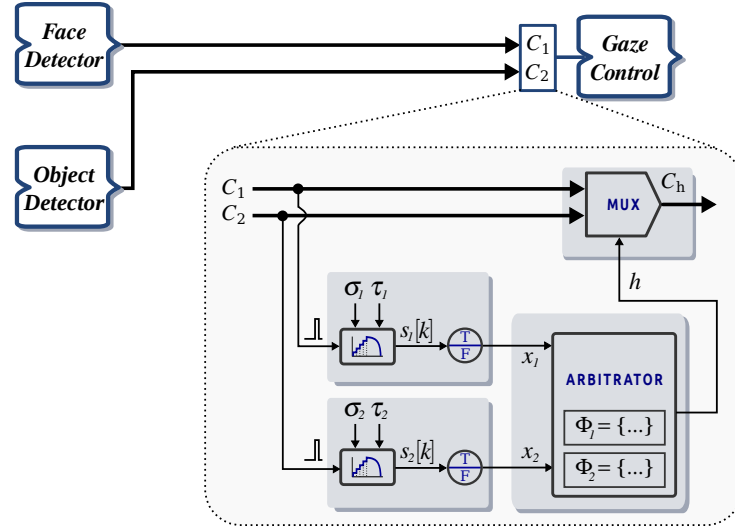


Figure 3.1: Two behaviors (The Face Detector and Object Detector) are connected to the input port of Gaze Control using connections  $C_1$  and  $C_2$ . The port arbitrator in Gaze Control coordinates the two behaviors using a set of rules and properties of each connection (dotted box). Please refer to the text for the definition of each symbol.

- Data are streamed out if and only if the behavior is active. For example, an object detector sends object position information through its output port only if the object has been detected.

We focus on the typical scenario of a publish–subscribe architecture in which modules (behaviors) can communicate asynchronously using connection points (ports). The key features we require are *i*) the output of a behavior can be connected to one or more input ports of other behaviors and *ii*) multiple outputs from different behaviors can be connected to the same input port of another behavior. For practical reasons we developed our architecture on top of the YARP middleware [1].

In the example from Figure 3.1, Face Detector and Object Detector can both send 3D position information to Gaze Control which controls the robot’s head to gaze accordingly. Behaviors run in parallel and can be distributed over a cluster of computers that communicate through network interfaces. Since there is no synchronization among behaviors, data can be delivered to an input port at any time, potentially causing con-

flicts. For example, in a simple scenario where a person keeps an object in front of the robot, Face Detector and Object Detector compete to get data to Gaze Control. An appropriate coordination mechanism avoids conflicts and, at the same time, it allows obtaining different behaviors (looking at a face or looking at the object).

We propose an arbitration mechanism between multiple, competitive connections to the input port of a behavior. As an example Figure 3.1 illustrates the arbitration in the case of a port with two connection (here, Face Detector and Object Detector are competitive connections to Gaze Control)\*. Messages arrive to a port from different channels (connections) and generate events. Arbitration happens in three stages: computation of the activation values, evaluation of the rules and selection. First, events are accumulated with leaky integrators to produce stimulation values for all connections. A connection becomes active when its stimulation level reaches a certain threshold. In the second stage, the port arbitrator selects a single connection among the ones that are active by evaluating a set of rules (i.e., written in first order logic) associated to each connection. This “winner” connection delivers data to the behavior whereas data from the other connections gets discarded.

As it is shown in Figure 3.1, the port arbitrator is implemented as a multiplexer that let, at most, one active connection deliver its data to the component at any given time. Here, we describe the parameters of each connection. In the following sections, we demonstrate how these parameters can be used to properly arbitrate multiple connections. Each connection  $C_i$  has the following parameters:

$$\Psi_i = \langle \sigma_i, \tau_i, \Phi_i \rangle, \quad i \in \{1 \dots m\}$$

$\Psi_i$  is a list of the properties of the  $i^{\text{th}}$  connection (identified by  $C_i$ ) to an input port with  $m$  connections;  $\sigma_i$  is the stimulation gain and  $\tau_i$  is the damping time.  $\Phi_i$  is the selection rule associated to the connection  $C_i$ . In the following sections, we explain how the selection rules are represented in first order logic and evaluated based on the activation state of the connections.

---

\*Notice that several models for the control of attention have been proposed in the literature that are much more appropriate for this specific task. However, we use the control of gaze as an example to demonstrate and validate the proposed mechanism.

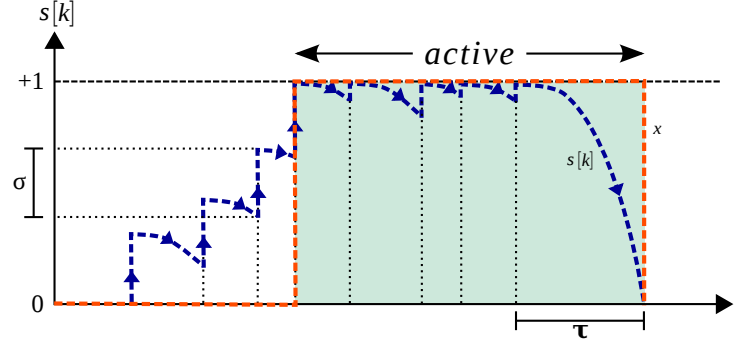


Figure 3.2: Computation of the stimulation and activation levels for each connection.  $s[k]$  is the stimulation level at time  $t_k$ .  $\sigma$  is the stimulation gain and  $\tau$  is the damping time. Stimulation is accumulated every time a new message arrives and it continuously decays over time. The connection is in active state once the cumulative stimulation reaches a threshold and until it gets completely discharged.

### 3.3.1 Computation of the activation values

Figure 3.2 illustrates how the activation value of each connection is computed. Arbitration cycles and data delivery happen at discrete events in time and are triggered whenever new messages arrive at the port from any connection  $C_i$ . Time values  $t_k$ ,  $k = 0, 1, 2 \dots$ , are associated to these discrete events using an internal clock. There is a stimulation level  $s_i[k]$  (at time  $t_k$ ) associated to every connection  $C_i$ ,  $i \in \{1 \dots m\}$ . All  $m$  stimulation levels in the port are updated at every instant  $k$  according to an exponential decay rule. Additionally, the stimulation level  $s_l[k]$  that corresponds to the channel  $l$  which has received the message, is increased by summing the corresponding stimulation gain  $\sigma_l$ , formally:

$$\begin{cases} [l]s_i[k-1] \cdot \left(1 - e^{-\frac{\lambda(t_k - t_{k-1} - \tau_i)}{\tau_i}}\right) & \forall i \in \{1 \dots m, i \neq l\}, \\ s_l[k-1] \cdot \left(1 - e^{-\frac{\lambda(t_k - t_{k-1} - \tau_i)}{\tau_i}}\right) + \sigma_i & i = l \end{cases} \quad (3.1)$$

where  $\lambda$  and  $\tau_i$  define the decay constants of the exponential function. Next,  $s_i$  is saturated to be within  $[0, 1]$ . Equation (3.1) formulates the calculation of  $s_i[k]$ . When  $s_i[k]$  reaches the threshold 1.0, the connection  $C_i$  is in *active* state until it gets completely

discharged and decays to zero. To simplify the notation, we drop dependence from  $k$  and define the activation value  $x_i$  as:

$$x_i = \begin{cases} \mathbf{true} & \text{if } C_i \text{ is active,} \\ \mathbf{false} & \text{otherwise.} \end{cases} \quad (3.2)$$

### 3.3.2 Representation of the rules

As we mentioned above, an active connection  $C_i$  (i.e., its  $x_i = \mathbf{true}$ ) has the opportunity to be selected by the arbitrator based on the selection rule specified by  $\Phi_i$ . The rule simply provides the necessary constraint for the selection of  $C_i$  in term of activation values  $(x_i \dots x_m)$  \*.

Let see how different selection rules can implement different behaviors. In the example from Figure 3.1 the robot will gaze at the face when data from connection  $C_1$  of Face Detector is selected by the arbitrator at Gaze Control. To gaze at a detected object, on the other hand, the port arbitrator should select the connection  $C_2$  so that object position data from Object Detector is delivered to Gaze Control. In the same example, suppose that we want the robot to track an object (continuously gaze at the object) when it appears in the view of the robot. This means that connection  $C_2$  should be selected when it is in active state (i.e.,  $x_2 = \mathbf{true}$ ), formally:

$$\Phi_2 = x_2$$

Imagine now we also want to track the face of a person, but only if there is no object in the scene. In other words, connection  $C_1$  should be selected if active, but only if  $C_2$  is NOT active. Therefore the corresponding rule should be specified for  $C_1$  and added to the arbitrator:

$$\Phi_1 = x_1 \wedge \neg x_2$$

In another scenario, suppose that we want the robot to gaze at an object if there is also a person in the scene, in terms of connections that means that the connection  $C_2$  should

---

\*A rule consistency validation is performed during the design time to ensure the rules specified in the port arbitrators does not contain contradiction.

be selected if both  $C_1$  and  $C_2$  are active:

$$\Phi_2 = x_2 \wedge x_1$$

In this case, we are not interested in tracking the face. Therefore we add the following rule:

$$\Phi_1 = \text{false}$$

That specifies that data from connection  $C_1$  is never delivered to the Gaze Control.

### 3.3.3 Selection Mechanism

Using the above equations, the selection mechanism is straightforward. When data arrives from connection  $C_i$  to an input port, the corresponding arbitrator has to decide whether to accept or discard it. First, using equations 3.1 and 3.2 the activation values for all connections ( $x_i \dots x_m$ ) are updated. The rule specified in  $\Phi_i$  is structured in a Binary Decision Diagram (BDD) [18]. Then, the arbitrator evaluates the rule and, if the constraints specified by the rule are satisfied, the index  $i$  of the current connection is given to the multiplexer, which in turn opens the corresponding channel to deliver data from  $C_i$  to the behavior (in Figure 3.1, this “winner” connection is indicated by  $h$ ). Otherwise the data are discarded. Notice that when the rules  $\Phi_i$  are specified for all  $m$  connections in the arbitrator, a consistency check ensures that only a single connection can be active at any given time.

## 3.4 Parameters tuning and design policy

The selection parameters can be used to further tune the behavior of the system. The stimulation gain  $\sigma$  and the damping time  $\tau$  are used together to control the reactivity of the system to external events. As an example, consider a behavior that checks the sensors of the fingertips of a robot and sends events whenever the robot touches an object. If these events are used by another behavior for collision avoidance in a safety context, a higher value for  $\sigma$  should be chosen to obtain a prompt reactive behavior.

Alternatively, if the output is used to stimulate a less time-critical behavior (i.e., a grasping action), a smaller value of  $\sigma$  is preferable to collect enough evidence from the fingers before grasping the object. The stimulation gain  $\sigma$  should be chosen in combination with the damping time  $\tau$ . Clearly, if  $\sigma_i$  and the stimulation rate (the frequency with which data arrives at the channel  $i$ ) are small with respect to  $\tau_i$ , the stimulation level  $S_i$  can never reach the threshold.

### 3.5 Experimental validation

In the following section, we present an experiment with the iCub humanoid robot. The main goal of the experiment is to demonstrate that our port-arbitrated mechanism allows for i) coordinating different distributed behaviors which compete to control the robot's actuators, ii) breaking down a complex system in subsystems that are configured, tested individually and finally combined together and iii) implementing a system that is reactive to the changes in the environment. We call this behavior "Take and return". The robot should perform a series of actions: (A) look for an object, (B) reach for the object, (C) grasp the object, (D) look for a person, (E) approach the person and (F) release (return) the object.

#### 3.5.1 First experiment: Take an object

In the first experiment we combine some simple behaviors to build a system that allows the robot to take an object. The user shows a known object to iCub and the robot tracks

Table 3.1: Arbitration rules for "Take an Object".

	Arm Control	Gaze Control	Hand Control
$\Phi_1$	$x_1 \wedge \neg x_2$	-	-
$\Phi_2$	<b>false</b>	-	-
$\Phi_3$	-	$x_3 \wedge \neg x_4$	-
$\Phi_4$	-	<b>false</b>	-
$\Phi_5$	-	-	$x_5$

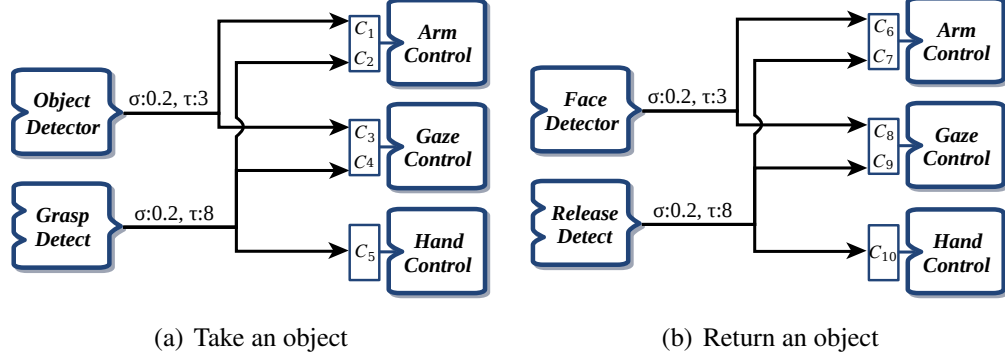


Figure 3.3: Configuration of the behavior network for “Take an object” and “Return an object”.

it with the eyes and grasps it with the hand. Figure 3.3(a) shows the behaviors and the configuration of the connections. For the sake of brevity, non-arbitrated connections are not shown in the figure (e.g., camera inputs). Object Detector receives streamed image frames from the robot cameras and produces the 3D position of the object when it is detected. Gaze Control and Arm Control receive a 3D position in the robot root frame and respectively control the head of the robot to gaze at the target and move the hand of the robot to the target position. Grasp Detect monitors the positions of the object and the hand to determine when they are close enough and issue a request to grasp. Hand Control controls opening and closing of the hand upon receiving release or grasp command.

As shown in Figure 3.3(a), the output of Object Detector is connected to Arm Control and Gaze Control which causes the robot to track and attempt to reach for the object.

Table 3.2: Arbitration rules for “Return an Object”

	Arm Control	Gaze Control	Hand Control
$\Phi_6$	$x_6 \wedge \neg x_7$	-	-
$\Phi_7$	<b>false</b>	-	-
$\Phi_8$	-	$x_8 \wedge \neg x_9$	-
$\Phi_9$	-	<b>false</b>	-
$\Phi_{10}$	-	-	$x_{10}$

The stimulation level  $\sigma = 0.2$  and damping time  $\tau = 3$  indicate that tracking and reaching should be started if there are enough events from Object Detector (i.e., at least 5 events within 3 seconds). Simultaneously, Grasp Detect checks if the object is graspable and if this is the case, it generates a grasp command to Hand Control. Table 3.1 shows the arbitration rules associated to each connection from Figure 3.3(a).

Constraint  $\Phi_1 = x_1 \wedge \neg x_2$  implies that data from connection  $C_1$  should be selected if  $C_2$  is inactive.  $\Phi_2 = \mathbf{false}$  specifies that data from Grasp Detect will be never delivered to the Arm Control. The same rules are applied for  $C_3$  and  $C_4$  in the arbitrator of Gaze Control. These are used to inhibit commands from Object Detector and therefore prevent the motion of the hand and head of the robot while grasping the object.  $\Phi_5 = x_5$  implies that grasp commands from Grasp Detect can be delivered to the Hand Control whenever the object is graspable.

### 3.5.2 Second experiment: Return an object

In the second experiment, we build another network of behaviors that allows the robot to return an object to the user (assuming it has grasped it). The configuration of the network is very similar to the previous experiment. As can be seen in Figure 3.3(b) Face Detector, now searches for a human face in the images and, when successful, it provides its 3D position to Arm Control and Gaze Control. That causes the robot to track the face and to extend the arm towards it. Release Detect, generates release commands to Hand Control if the hand is pointing toward the face, causing the robot to release the object. Table 3.2 shows the required arbitration rules to implement the scenario. Similar to the “Take an object” scenario, during release of the object, Release Detect inhibits the movements of the arm and the head. Notice that for simplicity, this behavior assumes that the robot has grasped the object. The module will be added in the next section that explicitly checks this condition.

### 3.5.3 Third experiment: Take and return

In the final experiment (“Take and return”), we exploit the behaviors implemented previously for “Take an object” and “Return an object”. Since we want the robot to



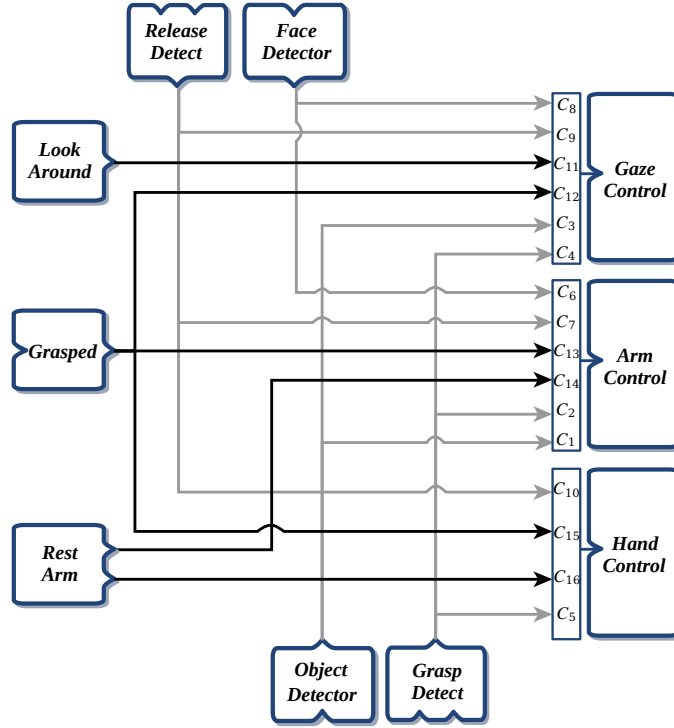


Figure 3.4: Configuration of behaviors for “Take and Return”. The connections that were previously used in “Take an object” and “Return an object” are shown here in gray.

return the object only if it has previously grasped it, we first introduce a new behavior, *Grasped*, which combines the information from the touch sensors and the hand encoders to produce a status message when the fingers are closed and the presence of an object is detected in the hand. The output of *Grasped* should inhibit all the connections in “Take an object” and enable “Return an object”. Figure 3.4 shows the configuration of behaviors. The connections that were previously used in “Take an object” and “Return an object” are shown here in gray. That emphasizes the fact that it is possible to build complex behaviors incrementally using existing subsystems without modifications. This is done by adding constraints to the arbitration rules of the subsystems. For example, in Figure 3.4, we want to inhibit tracking when the robot is holding the object. That is achieved by adding the constraints “ $\neg x_{12}$ ” to  $\Phi_3$  in the subsystem “Take an object”. This has the effect of inhibiting data from Object Detector (i.e.,  $C_3$ ) when

Table 3.3: Arbitration rules for “Take and Return”

	Arm Control	Gaze Control	Hand Control
$\Phi_1$	$(x_1 \wedge \neg x_2) \wedge \neg x_{13}$	-	-
$\Phi_2$	<b>false</b>	-	-
$\Phi_3$	-	$(x_3 \wedge \neg x_4) \wedge \neg x_{12}$	-
$\Phi_4$	-	<b>false</b>	-
$\Phi_5$	-	-	$x_5 \wedge \neg x_{15}$
$\Phi_6$	$(x_6 \wedge \neg x_7) \wedge x_{13}$	-	-
$\Phi_7$	<b>false</b>	-	-
$\Phi_8$	-	$(x_8 \wedge \neg x_9) \wedge x_{12}$	-
$\Phi_9$	-	<b>false</b>	-
$\Phi_{10}$	-	-	$x_{10} \wedge x_{15}$
$\Phi_{11}$	-	$\neg(x_3 \vee x_4 \vee x_8 \vee x_9)$	-
$\Phi_{12}$	-	<b>false</b>	-
$\Phi_{13}$	<b>false</b>	-	-
$\Phi_{14}$	$\neg(x_1 \vee x_2 \vee x_6 \vee x_7)$	-	-
$\Phi_{15}$	-	-	<b>false</b>
$\Phi_{16}$	-	-	$\neg(x_5 \vee x_{10})$

the output of Grasped (i.e.,  $C_{12}$ ) is active. The necessary rules for inhibiting other connections to Arm Control and Hand Control in “Take an object”, are added similarly. Notice that arbitration rules are added and removed by specifying connection parameters to the port arbitrators and without changing the code implementing the individual modules. Table 3.3 represents the full list of required rules to implement “Take and return” using behaviors from Figure 3.4.

We now add modules to put the arm in a resting position and randomly look around in search for the object. The module Look Around sporadically sends random position commands to Gaze Control in search for objects or faces. This behavior must be clearly inhibited while “Take an object” and “Return an object” are active. To do so, we add the necessary constraints to the arbitration rule  $\Phi_{11}$  of connection  $C_{11}$  (i.e.,  $\Phi_{11} = \neg(x_3 \vee x_4 \vee x_8 \vee x_9)$ ). We also add the module Rest Arm that attempts to move the arm to a predefined resting position by periodically sending release commands

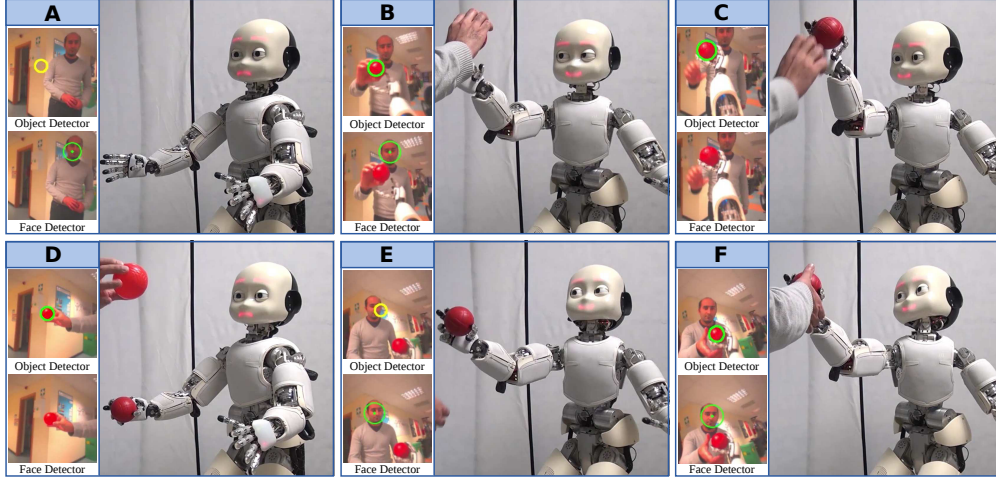


Figure 3.5: iCub performing the “Take and return” experiment, **A**: looking for the object, **B**: reaching for the object, **C**: grasping the object, **D**: looking for a person while holding the object, **E**: approaching the person and **F**: releasing the object.

and arm–resting–position commands to Arm Control and Hand Control. Appropriate constraints are added to  $\Phi_{14}$  and  $\Phi_{16}$  to inhibit that behavior when the robot is taking or returning the object.

The arbitration mechanism was implemented using YARP [54] ports. The “Take and return” behavior was then tested on the iCub robot (see Figure 3.5). Figure 3.6 plots the stimulation values of all the connections to Arm Control during the experiment. Stimulation is plotted in green when the corresponding connection is selected and in orange otherwise. At the beginning and before  $t = 140$ , the hand of the robot is empty (Grasped is inactive) and Rest Arm can command Arm Control and Hand Control, to keep the robot arm in a resting position. That corresponds to the behavior A in which the robot is looking for the object. Notice that, at the same time, Face Detector is also activated (a person enters the scene); however since the selection of Face Detector depends on the output of Grasped (i.e.,  $\Phi_6 = (x_6 \wedge \neg x_7) \wedge x_{13}$ ), the required constraints are not satisfied and data from Face Detector are not delivered to Arm Control. At  $t = 140$ , the person shows the object to the robot; this increases the simulation level of Object Detector to activation and the robot reaches for the object (behavior B). At  $t = 142$ , Grasp Detect is stimulated; this prevents the robot’s arm from moving and at

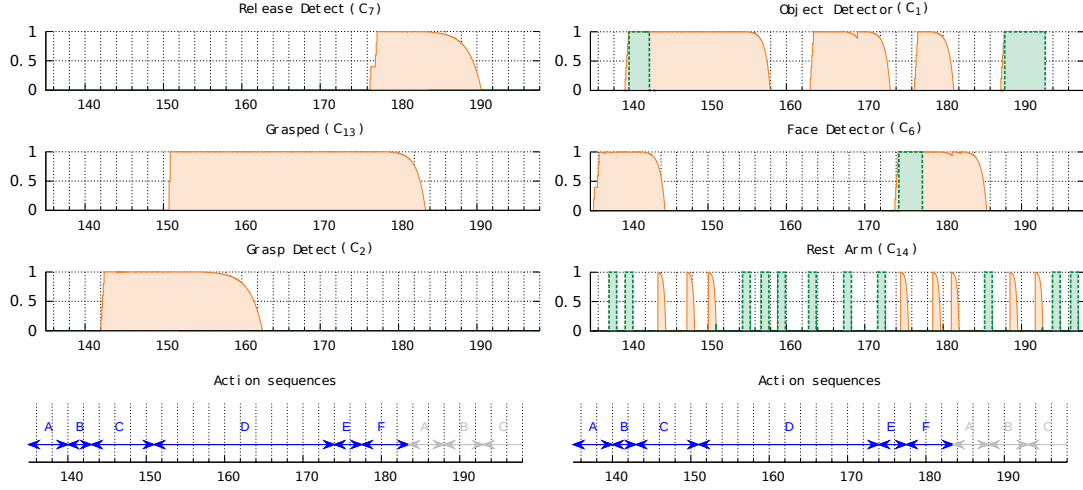


Figure 3.6: Arbitration in Arm Control during the “Take and return” experiment. The plots show the values of  $S_i[k]$  and the selected connection by the arbitrator of all the  $m$  input connections from the modules Release Detect, Grasped, Grasp Detect, Object Detector, Face Detector and Rest Arm (see Figure 3.4). Horizontal axis is time  $t$  and vertical axis represents stimulation values  $S_i[k]$ . Stimulation is plotted in green when the corresponding connection is selected and in orange otherwise. The bottom graph shows the action of the resulting behavior, i.e., **A**: looking for an object, **B**: reaching for the object, **C**: grasping, **D**: looking for a person, **E**: approaching a person and **F**: releasing the object.

the same time sends grasp commands to Hand Control. That corresponds to behavior C in which the robot grasps the object. Notice that, during B and C, Rest Arm is also inhibited. At this point, the robot holds the object (behavior D) and Grasped is active. The robot cannot reach for another object. That can be noticed in the plot of the activation value  $C_1$  that corresponds to the module Object Detector: the latter is stimulated but its output gets inhibited by Grasped. At  $t = 174$ , the person enters again in the scene. Face detector is now activated and can be selected by the arbitrator because Grasped is active. That makes the robot move the hand towards the face (behavior E). At  $t = 178$  Release Detect gets activated; this causes the robot to release the object (behavior F). It also inhibits commands from Face detector and Rest Arm.

During the “Take and return” experiment, the behavior of the robot was tested under

different conditions (e.g., by showing a face before grasping or by showing the robot another object after grasping). These tests demonstrated that the overall behavior is intrinsically reactive to the environment. In fact, all behaviors continuously monitor the conditions under which they are activated. Unexpected situations are thus automatically handled by the network of behaviors, even if they were not explicitly considered at design time. A particular situation in the experiment demonstrated this explicitly. While the robot was returning the object, the user decided to anticipate the robot and took the object directly from the hand before the reaching command was completed. As a consequence, the stimulation of Grasped decreased and prevented all behaviors in “Return an object” to run. Finally, the output of Rest Arm was no longer inhibited and could command the arm to go back to the initial state. Therefore, the system went back to the initial state (A).

## 3.6 Conclusions

In this chapter, we have introduced an arbitration mechanism for a network of behaviors based on port arbitration. We have shown that our approach allows to implement a non-trivial behavior that involves a sequence of actions. Remarkably, we have shown that the final behavior can be incrementally built as a composition of existing, simpler behaviors. Our approach is also fully distributed and minimizes the additional links required to perform arbitration. We tested the behavior in different conditions and demonstrated that the resulting behavior is intrinsically robust and reactive to unexpected changes in the environment. More importantly, since no explicit modules are required to manage the coordination, no task-dependent code was written to implement the final behavior which as a result was exclusively built out of re-usable modules.

## **Chapter 4**

# **Modeling robotic behaviors using port arbitration in YARP**

### **4.1 Introduction**

Robotic researchers are developing increasingly complex humanoid robots which are aimed to be employed in unstructured and dynamical environments. Programming such robots to achieve robust task execution in a dynamical environment is a daunting task and requires dealing with many uncertainties and changes in the environment. Behavior-based approaches have particular interest due to its fast response to external events. Traditionally it has been used in robotic applications in which reactivity is crucial but it has also shown scalability problem when the number of the behaviors increase. This, practically, has limited its application domain to low-level control systems and the subfield of mobile robotics [8, 86].

In chapter 3 we have demonstrated an arbitration mechanism for a network of behaviors based on port arbitration. We have also shown that using our approach, how a complex application can be decomposed into smaller subsystems which are configured and tested individually and, finally, combined together to implement the desired behavior. However, the approach imposes certain limitations. First, it requires setting a certain number of parameters and rules. In the examples described in section 3.5, that was done quite intuitively; however, this may not scale well with the number of

modules and the complexity of the behavior. Second, since coordination is completely distributed, it is also more difficult to characterize the current state of the system and detect or monitor the behavior of the system as a whole. However, this information is present in the system and it can be made available by advertising the connection parameters.

To facilitate formulating the arbitration rules, we introduce a mechanism for modeling and coordination of behaviors based on port arbitration. The main goal of the work presented here is to build an architecture for behavior-based robot programming while keeping interesting features of the classical behavior-based systems. Coordination between modules is achieved by defining a set of rules that specify how to arbitrate conflicts between modules that run concurrently and compete for the same resources. The first step of our approach is to describe the software components and develop a behavioral representation of the task. The latter is then used to extract the required rules that allow coordinating of the software components to achieve the desired behavior. We implemented our approach using the YARP middleware [54] and tested it by developing a complex behavior on the iCub humanoid robot [53]. We show that using our approach we could implement a task that involves a sequence of actions. More importantly we demonstrate how, based on different behavioral descriptions, the same software components can be reused in different applications.

## 4.2 Related Work

Behavior-based systems (BBSs) have been devised to program robot applications that do not rely on models of the environment and for which reaction to sensory feedback is crucial. However, BBSs are difficult to design when the task involves a large number of modules and connections carrying heterogeneous data. Perhaps, this is one of the reasons why behavior-based approaches are not widely applied to the humanoid robot applications. The crucial problem is to represent behaviors and components separately so that the latter can be reused more freely. In modern robotic software middlewares coordination is more difficult because components run asynchronously and are distributed across a network of computers. Generally, these problems are not well

addressed by existing frameworks. For example, iB2C [70] limits behaviors to be implemented as a single component, DBN [44] does not allow behaviors to be distributed and parallelism is simulated in iterations, whereas DAMN [73] forces behaviors to be coordinated using a centralized coordinator.

Best practices in robotics [15] promotes the idea that composition and coordination of software component should be separated during the life-cycle of software development. Nicolescu et al. [61] also emphasize that a proper abstract representation of the behaviors is crucial for the development of complex robotic application. Our behavior-based framework provides support for both these key features. In our approach, building an application out of reusable software is done in two phases. First, software components are configured and interconnected in the (distributed) system. Second, a behavioral model is developed which describes the desired behavior of the system. Coordination is then defined by extracting a set of rules from the behavioral model. These rules determine how data is allowed to travel across the network of components and therefore implicitly define which components are inhibited or free to run. Therefore, based on different behavioral description, the same software components can be reused to implement different robotic applications.

### 4.3 Representing arbitration rules using named connections

In the previous chapter, we have demonstrated the concept of port arbitration and how the arbitration constraint can be represented based on the activation status of each connection. In the following section, we concentrate on the representation of arbitration rules in YARP framework where the connection can be described as a pair of named source and destination ports.

In the example from Figure 4.1, Object Detector is a component which processes the streamed images from the robot camera and produces the 3D position of the object when detected. Its output is connected to Gaze Control which receives a 3D position in the robot root frame and controls the head of the robot to gaze at the target point. The output port of Object Detector is also connected to the Arm Control component which



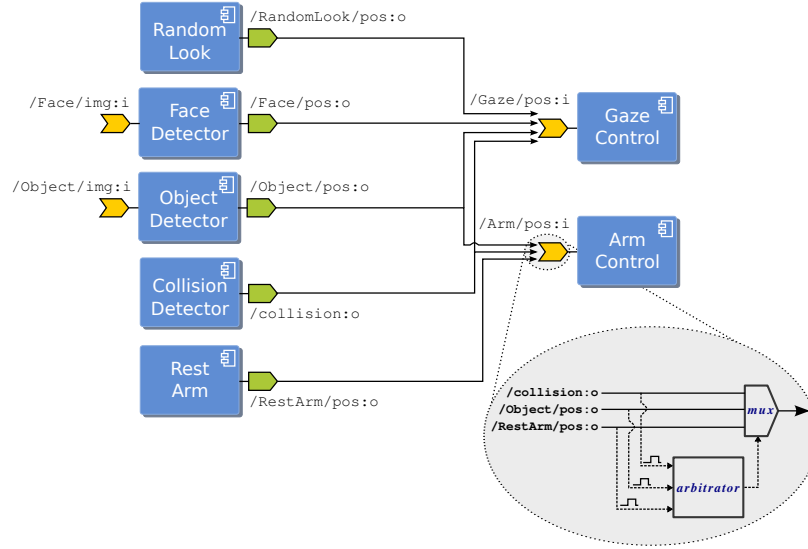


Figure 4.1: An example of different YARP components and the connections among them. The shaded box represents how an arbitrator is employed in the input port of a component to arbitrate between multiple connections to the same port.

moves the robot arm's end-effector to reach the target point received from its input port. A crucial aspect is that multiple outputs can be connected to the input port of a component. Without proper coordination among the components, data from different components can be delivered to an input port at any time, potentially causing conflicts. To solve this problem, as we have explained before, every input port has an arbitrator which can be configured with a set of rules to properly arbitrate the data received from multiple sources. We propose a mechanism to describe the behavioral model of the task. This model allows to derive the necessary rules that properly configure the arbitrators to implement the task. We define the ingredients of our port-arbitrated coordination system as follow:

- A pair of source and destination names identifies the *connection* from an input port to an output port (e.g. {/Face/pos:o, /Gaze/pos:i}).
- An *active* connection is a connection which has recently delivered data. When data arrives to an input port from a connection, the latter becomes active and remains active for a constant time  $T$ . The connection will be inactive if no more

data arrives within time  $T$ . Notice that the activation of a connection is defined solely in terms of the data it delivers to the port and irrespectively of the result of the arbitration.

Each input port has an arbitrator which selects a single connection among the ones that are active at each time. This concept is also illustrated in Figure 4.1 for the case of three connections. The Rest Arm periodically sends the resting position of the arm through `RestArm/pos:o` to Arm Control. This causes the robot to park and keep the arm in the resting position. To grasp an object we want to hand over control of the arm to another component (i.e., Object Detector) that sends the position of the object to be grasped to Arm Control. This can be done by inhibiting the connection from `/RestArm/pos:o` to `/Arm/pos:i` in the arbitrator of `/Arm/pos:i`. In other words we want to specify a rule so that the connection  $\{\text{/RestArm/pos:o}, \text{/Arm/pos:i}\}$  can be selected only if connection  $\{\text{/Object/pos:o}, \text{/Arm/pos:i}\}$  is inactive, formally:

$$\text{/RestArm/pos:o} \wedge \neg \text{/Object/pos:o} \Rightarrow \text{Select}(\text{/RestArm/pos:o})$$

Suppose now we add another component which is responsible for stopping the the arm upon collision. This component is called Collision Detector in Figure 4.1 and it sends status messages through the port `/collision:o` when it detects that the arm collides with an object (e.g. using tactile or torque sensors). The desired behavior can be achieved by adding rules in `/Arm/pos:i` so that activation of `/RestArm/pos:o` and `/Object/pos:o` is inhibited by `/collision:o`, i.e.:

$$\begin{aligned} & \text{/RestArm/pos:o} \wedge \neg \text{/Object/pos:o} \wedge \neg \text{/collision:o} \\ & \Rightarrow \text{Select}(\text{/RestArm/pos:o}) \end{aligned}$$

$$\text{/Object/pos:o} \wedge \neg \text{/collision:o} \Rightarrow \text{Select}(\text{/Object/pos:o})$$

Notice that if no rules specifically select a connection, the latter can never deliver data to the component. In our example, no rule is written for `/collision:o` in Arm Control; so Collision Detector will never deliver data to this component. As it is, its activation state is only used in the evaluation of the rules of other connections.

## 4.4 Modeling the behaviors

In general, behavior-based robotic controllers consist in a collection of behaviors and are implemented as control laws to achieve and/or maintain goals [52]. In our approach, behaviors can be described as a set of rules for the port arbitrators. In Figure 4.1, Face Detector sends the position of the detected face to Gaze Control to follow it. In other words, to implement a behavior called Follow Face, the connection `/Face/pos:o` to `/Gaze/pos:i` should exist and be selected by the port arbitrator in `/Gaze/pos:i`. To implement another behavior we call Look Around, the connection `{/RandomLook/pos:o, /Gaze/pos:i}` should be selected by the port arbitrator in `/Gaze/pos:i` to deliver the random position data generated by Random Look to the Gaze Control. Desired behaviors, therefore, can be implemented by selecting connections which are required to deliver data among specific components. At the behavioral description level, we concentrate only on the connections among the components and the necessary rules that select these connections in the arbitrators. The rules can be provided by specifying *configuration* of the connections required for implementing the behavior, under which *condition* the behavior can be activated and the list of behaviors it should *inhibit*.

In Figure 4.1 we have shown an example of the composition of some components and their connections. Based on them, in Figure 4.2 we depict an example of behavioral description for a task in which the robot searches around, follows human's faces and tracks an object with the hand. In the figure, *Follow Face*, *Look Around*, *Rest Arm* and *Track Object* represent behaviors. *Follow Face* and *Look Around* are grouped together to describe, by composition, another (meta behavior) behavior called *Be Curious*. *Be Curious* implements a behavior that let the robot randomly look around or follow a human's face if a person appears in the scene. The red arrow from *Follow Face* to *Look around* gives higher priority to the first behavior whose activation inhibits the second. *Rest Arm* describes a behavior that keeps the arm in the resting position. *Track Object* implements tracking of an object with the gaze and reaching for it with the hand. It also inhibits *Rest Arm* and the meta behavior *Be Curious* to prevent them from interfering during tracking. The condition " $\neg$  /collision" in *Track Object* implies that robot can track an object only in absence of collisions. *Track Object*, *Rest Arm*

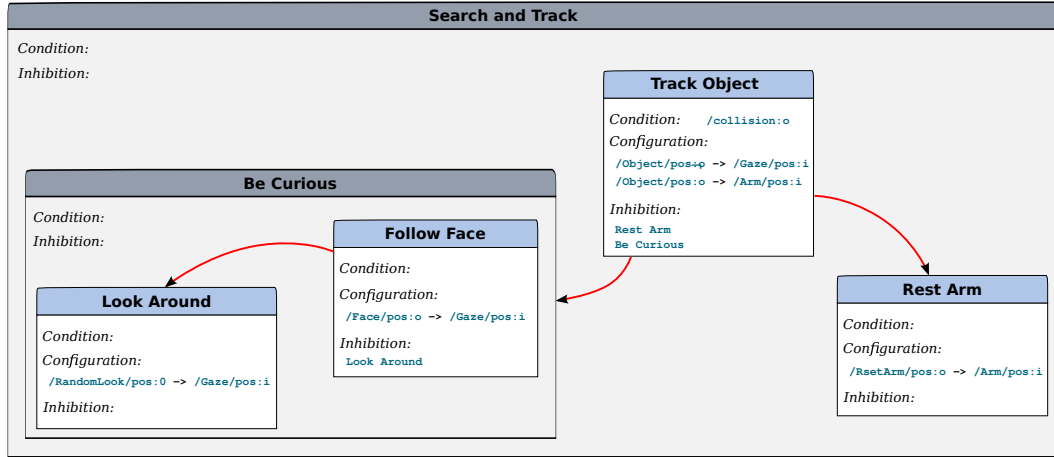


Figure 4.2: An example of the behavioral model that uses the components from Figure 4.1 to implement the Search-and-Track behavior. This behavior allows the robot to look around in search for a face or an object. When the robot detects a face it tracks it with the gaze. When it detects an objects it follows it with the gaze and reaches for it. The overall behavior is implemented by coordinating simpler behaviors. Correct coordination is implemented by inhibitions among behaviors (red arrows). See also the description in the text.

and *Be Curious* are further grouped to describe another meta behavior we called *Search and Track*.

#### 4.4.1 Behavior Specification

A behavior (or a meta behavior) has the following properties:

*Configuration* of a behavior is the list of connections which should be selected by the port arbitrators to implement the behavior. For meta-behaviors, configuration is as a list of behaviors or other meta-behaviors. For example, in Figure 4.2, the configuration property of *Track Object* implies that to follow an object with the head, `/Object/pos:o` should feed data to Gaze Control at `/Gaze/pos:i`. Tracking with the hand is achieved by sending `/Object/pos:o` to Arm Control at `/Arm/pos:i`. Notice that here we focus only on the connections which define the behavior of the system, but

other connections are required for proper functioning of some modules (e.g. Face Detector and Object Detector require connections from the robot cameras) For simplicity we do not consider these connections here.

*Condition* is an optional property which specifies in, first-order logic, a constraint that should be verified for the behavior to be activated. The condition  $\neg /collision:o$  of *Track Object* requires that all the connections specified in its Configuration should be selected only if the port */collision:o* is inactive (i.e. it is not sending messages). In a meta-behavior the Condition affects all its child behaviors, i.e. conditions from all parent meta-behaviors in a hierarchy are conjuncted and inherited by all child behaviors.

*Inhibition*, specifies inhibitions between behaviors or meta-behaviors. Specifying inhibitions allows coordinating behaviors that are competing for the same resources. In Figure 4.2 we define the behavior *Look Around* which is implemented by connecting ports of Random Look to Gaze Control. We also define *Follow Face* and *Track Object*. These behaviors compete to control the gaze of the robot by sending commands to Gaze Control at */Gaze/pos:i*. Conflicts are avoided by further specifying the overall behavior of the robot and assigning inhibitions. In Figure 4.2 *Follow Face* inhibits *Look Around*. In more details this tells the arbitrator in */Gaze/pos:i* that connection  $\{ /RandomLook/pos:o, /Gaze/pos:i \}$  should not be selected when connection  $\{ /Face/pos:o, /Gaze/pos:i \}$  is active, because Face Detector is sending data to Gaze Control. A behavior can also inhibit a meta-behavior. In this case, the behavior inhibits all the behaviors in the meta-behavior. In Figure 4.2, *Track Object* inhibits *Be Curious*, i.e. it inhibits *Follow Face* and *Look around*. In practice this corresponds to assigning decreasing priorities to *Track Object*, *Follow Face* and *Look around* to avoid conflicts in Gaze Control. Similar rules are applied if a meta-behavior inhibits another behavior or another meta-behavior. For the sake of modularity and reusability, behaviors can only inhibit other behaviors within the same meta-behavior. For example in Figure 4.2, *Follow Face* is not allowed to inhibit *Rest Arm*.

## 4.5 Extracting Rules from Behavioral Model

In the previous section we have described how behaviors are modeled using connections between ports. In this section we explain how the necessary rules for the arbitrators are extracted from the behavioral model. Every behavior has a list of connections specified by its Configuration property. The properties Conditions and Inhibition determine an extra set of constraints that are applied to the port arbitrators of its connections.

For example in Figure 4.2, *Look Around* is inhibited by *Follow Face*. Both behaviors have a connection to `/Gaze/pos:i`; Thus the following rule is added to the port arbitrator in `/Gaze/pos:i`:

$$\text{RandomLook/pos:o} \wedge \neg \text{Face/pos:o} \Rightarrow \text{Select}(\text{RandomLook/pos:o})$$

*Look Around* is also inhibited by *Track Object* (through the inhibition to *Be Curious*). Therefore the previous rule is updated with “ $\neg \text{Object/pos:o}$ ” to reflect the new constraint:

$$\begin{aligned} & \text{RandomLook/pos:o} \wedge \neg \text{Face/pos:o} \\ & \wedge \neg \text{Object/pos:o} \Rightarrow \text{Select}(\text{RandomLook/pos:o}) \end{aligned}$$

A behavior’s Condition and the conditions that are inherited from the parent groups are also added in the same way to the port arbitrators of all of the connections specified in Configuration. For example, the constraint “ $\neg \text{collision:o}$ ” is added to the rules for `/Object/pos:o` in the arbitrators at `/Gaze/pos:i` and `/Arm/pos:i`.

To summarize, the algorithm to extract the arbitration rules from the behavior model is easily done in two steps for each behavior *i* in the model. First: the Condition of *i* is updated with all the conditions it inherits from the parent meta-behaviors. This condition is added as an extra constraint to the port arbitrators of all the connections specified in Configuration. Second: further conditions are extracted from all inhibitors of *i* and added to the rules of the corresponding port arbitrators.

## 4.6 Reference Implementation in YARP

The behavioral model described in the previous section can be represented using Extensible Markup Language (XML). Listings 4.1 illustrates the representation of *Be*

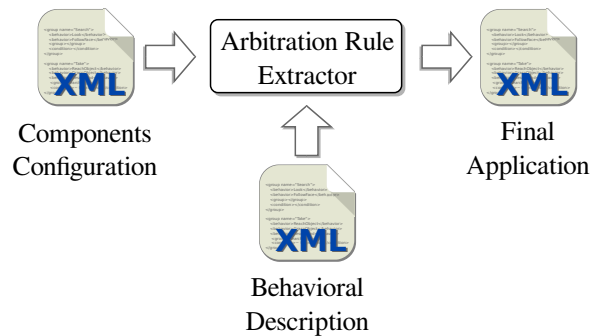


Figure 4.3: Application generation from behavioral description.

*Curious* behavior in XML format. The model are used by a third-party tool to extract the arbitration rules and update the configuration of connections. The concept is illustrated in Figure 4.3.

```

<define name="gaze"> /Gaze/pos:i </define>

<group name="Be Curious">
  <behavior>Look Around</behavior>
  <behavior>Follow Face</behavior>
  <condition></condition>
</group>

<behavior name="Look Around">
  <configuration at="{gaze}">/RandomLook/pos:o</configuration>
  <condition></condition>
  <inhibition></inhibition>
</behavior>

<behavior name="Follow Face">
  <configuration at="{gaze}">/Face/pos:o</configuration>
  <condition></condition>
  <inhibition>Look Around</inhibition>
</behavior>

```

Listing 4.1: The representation of "Be Curious" behavior in XML format.

YARP framework offers a way to describe the configuration of components and their connections in XML format which known as application description file (See Chap-

ter 7 for further information). In short, the application description file contains all the required modules, their configuration via parameters, the way they are interconnected, and the necessary information for their deployment. The required information for orchestration (coordination) of these modules can be represented in another XML file using our behavioral model based on port-arbitrated mechanism. The final application, thus, can be generated by the combination of these files using the third-party tool (i.e., 'yarpmanager'). Therefore, based on different behavioral model, the same components can be used to develop different applications.

## 4.7 Experimental Validation

To demonstrate the applicability of the approach, we refer to the similar “Take and return” experiment from previous chapter (Section 3.5) and re-implement it using our behavioral description \*. As we have explained before, The robot should perform the following actions in a series: (A) look for an object, (B) reach for the object, (C) grasp the object, (D) look for a person, (E) give the object to the person and (F) release the object<sup>†</sup>. We show that our behavioral model can be used to describe the behaviors and constraints to let the robot achieve the task. The necessary rules are extracted to configure the port arbitrators to properly coordinate all components. We used the components from Figure 4.1 (with the exception of Collision Detector) and add other components which are shortly described here:

- Hand Control, controls opening and closing of the robot’s hand upon receiving release or grasp commands at the input port /Hand/cmd:i.
- Grasp Detect, monitors the relative position of the object and the hand to determine when they are close enough and, when this happens, it issues a request to grasp through /Grasp/cmd:o.

---

\*The complete source code and the configuration files used to perform the experiment can be freely accessed at <https://svn.code.sf.net/p/robotcub/code/trunk/iCub/contrib/src/behaviorBased>

<sup>†</sup>To simplify, the robot releases the object when the hand is close to the person; this is achieved by monitoring the relative position of hand and the person’s face



- Release Detect, determines when the hand is close to the person and the object can be released. If this is the case, it generates release commands to the Hand Control through `/Release/cmd:o`.
- Open Hand, this module keeps the hand open by periodically sending release commands through `/OpenHand/cmd:o` to Arm Control.
- Grasp, combines sensory feedback from the hand to produce a status message through `/grasp:o` whenever the presence of an object is detected in the hand.

For the sake of brevity, the integration of the components and connections used in the experiment are not represented here. However they can be inferred from the behavior model shown in Figure 4.4 and from the Figure 3.4.

First we shortly review the behaviors which are used in the experiment and then we demonstrate how they are constrained to implement the “Take and return” scenario. In Figure 4.4, three meta-behaviors are defined: “*Rest and Search*”, “*Take*” and “*Return*”.

*Rest and Search* lets the robot keep the hand and arm in the resting position while randomly looking around. This is implemented using *Rest arm* and *Look Around*. *Open Hand* is constrained by “ $\neg$ /grasped:o”, this allows to hold the object in the hand after it has been grasped.

*Take* allows the robot to track and grasp the object by combining the behaviors *Track Object* and *Grasp Object*.

Within *Take*, *Grasp Object* inhibits *Track Object* to prevent moving the hand and the head while the robot is closing the object. *Take* is also subject to “ $\neg$ /grasped:o”. This means that this behavior is executed only if required (i.e. the hand is empty).

*Return* represents a behavior for returning the object to a person. It combines *Reach Face* to track with the gaze the face of a person and extend the arm towards it. *Release Object* checks if the hand of the robot close to the the person to release the object. *Release Object* inhibits *Reach Face* to maintain the robot stationary while it releases the object. Execution of *Return* is constrained by the condition “/grasped:o”. This means that this behavior is executed only when required, i.e. if the robot is holding the object in the hand.

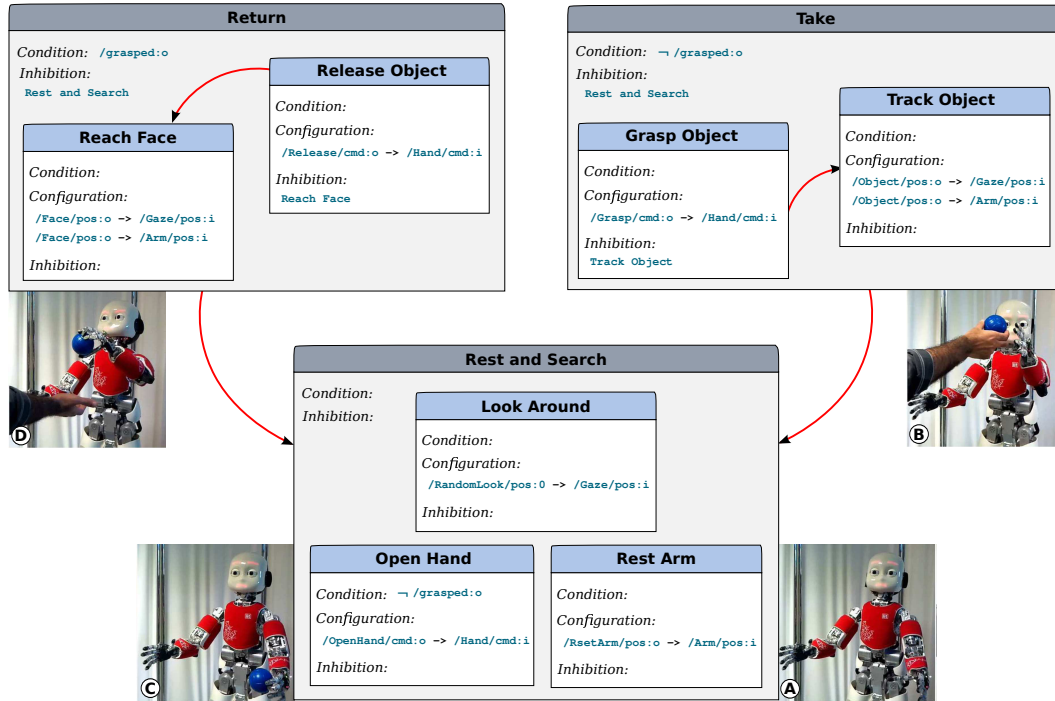


Figure 4.4: The description of the behavior “Take and Return” implemented on the iCub robot.

Inhibitions between the meta-behaviors allows the correct interaction. The resulting behavior alternates *Rest and Search*, *Take* and *Return*. It is important to point out that activation of the behaviors is dictated by the sensory feedback and the activation of the other modules, and, as such, is intrinsically reactive. This means that transitions between behaviors can happen at any time and do not follow a predefined, fixed order.

At the beginning of the experiment the hand of the robot is empty and no faces or objects are visible. *Take*, therefore, is idle. *Return* is inactive because “`/grasped : o`” is inactive (the component *Grasped* does not send messages). *Rest and Search* is not inhibited and can control the robot to look around while keeping the arm in the resting position and the hand open.

Then a user appears and shows the object to the robot. The Object Detector component, detects the object and streams out its position. *Take* is no longer idle because *Track Object* gets activated. *Take* inhibits *Rest and Search* and the robot starts tracking the

object. *Grasp Object* checks whether the robot hands is close to the object. If this is the case it gets activated and start sending commands. It stops the robot by inhibiting *Track Object* and it requests to grasp the object (through `/Hand/cmd:i`).

When the robot successfully grasps the object, the condition “ $\neg/\text{grasped:o}$ ” is no longer satisfied. *Take* does not run and no longer inhibits *Rest and Search*. Thus the robot starts looking around and keeps the hands in the resting position. The condition “ $\neg/\text{grasped:o}$ ” prevents *Open Hand* to become active and avoids that the object is dropped. On the contrary, the necessary condition in *Return* are satisfied. Thus when a person appears in the scene, *Reach Face* is activated and starts extending the hand toward the person. At the same time, Release Detector monitors if the hand points toward the person. When this happens, it starts sending release commands to `/Hand/cmd:i` through `Release/cmd:o` and the robot releases the object. While this happens *Release Object* inhibits *Reach Face* to maintain the arm stationary. During this process *Return* inhibits *Rest and Search*. After the robot has released the object, the condition “ $\neg/\text{grasped:o}$ ” is no longer satisfied and the behavior goes back to the initial state in which only *Rest and Search* is active.

The experiment is completely implemented using the described behavioral model. The correctness of the behavioral model and the extracted arbitration rules has been proven by comparing them with those which were manually developed for the same experiment in the previous chapter.

## 4.8 Conclusions

This chapter has introduced a mechanism based on port arbitration for modeling and coordination of behaviors. We have illustrated that our approach allows implementing a non-trivial behavior that involves a sequence of actions. We tested the behavior in different conditions and demonstrated that it is intrinsically robust to unexpected changes in the environment. We have shown how robotics tasks can be represented using our behavioral description model and coordinated in a distributed component-based framework without any central coordinator. Remarkably, We demonstrated that

in our framework, based on different behavioral descriptions, several robotic applications can be implemented using the same reusable software components.

## Chapter 5

# Extending Data Flow Port with Monitoring and Arbitration using Scripting Languages\*

### 5.1 Introduction

Writing a reusable software component requires a sense of taste. Two important choices to be made are how the component expresses its output, and what it expects of its input. With reusability in mind, there is pressure to be as generic as possible: to offer everything useful the component “knows” on the output, and to accept all sorts of variants on the input side. However, for any particular application, this generality is decidedly suboptimal. It can result in slow development and higher bandwidth requirements. The opposite approach is to let specific applications drive the development of the component; this may lead to faster development but can seriously limit component reusability. Indeed the “5C” paradigm [16], which is gaining popularity in robotics, dictates separation of concerns between Computation, Communication, Coordination, Configuration and Composition. Following this paradigm we introduce an approach where coordinating logic can be transparently inserted into a reusable com-

---

\*This chapter is based on Ali Paikan, Paul Fitzpatrick Giorgio Metta and Lorenzo Natale. Data Flow Port’s Monitoring and Arbitration. *Journal of Software Engineering for Robotics*.

ponent, along with data transforms unanticipated by the component author. With this approach, a robot's coordination system is no longer limited to passively receiving reports from far-flung components in their chosen formats and on their chosen schedule. It can now actively change how data is summarized and how it is communicated. This is achieved by using the component's common middleware as a hook to load arbitrary coordination logic into the external-facing interface of components.

### 5.1.1 Motivating Example

In order to clarify the main concern and contribution of this chapter, we consider a typical object tracking and reaching scenario, in which the robot is programmed to detect a moving object, to follow that object with its gaze, and to reach for it with its hand. Figure 5.1 illustrates the Data-flow architecture of the scenario. The image data from a pair of stereo cameras are given to two instances of Object Detector each computing the 2D position of the object in the camera frames. These modules feed this information to the 3D Position Estimator module, which performs the required geometric computations to calculate the position of the object in the robot frame, and finally sends those coordinates to the Head Control and Arm Control modules. The latter control the robot's head and arm respectively to look at the object and reach for it.

The overall behavior of the system can be fairly robust if every subsystem behaves as intended. However, some failures or uncertainties in the object detection or 3D position estimation can cause nondeterministic behavior of the robot. Klotzbucher et al. [45] characterize this as a typical coordination problem and propose having a lightweight coordination system using a state machine. The coordinator reacts to explicit events (e.g., events generated by the 3D Position Estimator if the object is not visible to the robot) and changes the state of the system so that an appropriate decision can be made, such as stopping the Arm Control module.

To increase the robustness of the coordinator and reusability of the subsystems (i.e., computational modules), Klotzbucher et al. also propose the Coordinator-Configurator pattern to separate the coordinator from the computational module. A pure coordinator requires to be informed via events about relevant changes in system state. Required

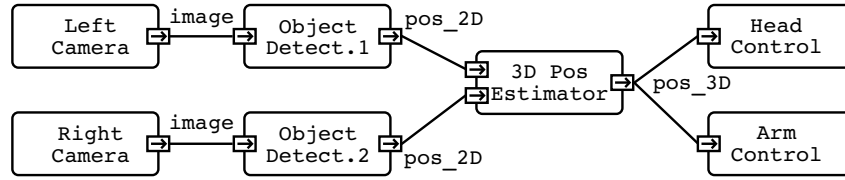


Figure 5.1: Data-flow architecture of an object tracking application

events for the coordinator can be generated in different ways. One way to achieve that is to extend the functionality of computational modules and configure them to raise the proper events. For example, 3D Position Estimator can generate different events when the object is not visible or the certainty of detected object drops below a configurable threshold. Klotzbucher et al. argue that this approach can be favorable if the constraint for generating event is computational dependent. On the other hand, if the constraint is application domain specific (e.g., a latency in communication between Object Detector and 3D Position Estimator), this approach severely limits the reusability of the computational module. Moreover, to reuse components with different coordination systems (e.g., event processing in BDI [89]), the required events by the coordinator should match with those generated by the computational modules. If this is not the case, the subsystem from one side should be modified.

Another approach is to introduce a separate component which remains between the computational module and the coordinator. The component can act as a monitor which communicates with the computational module and generates events for the coordinator. Alternatively, it can be used to translate existing events into the format which is required by the coordinator. That, in fact, requires implementing an application specific (likely not reusable) module and introduces additional communication and deployment overhead to the system which may not be acceptable in some distributed applications.

To overcome these shortcomings, we propose an approach which is a pragmatic compromise between reusability and performance. In our approach, components' data flow ports [76] are extended with scripting programming language capability. Using the scripting language, a monitor entity can be embedded in the output or input ports of component which monitors data and generates proper events for the coordinator. We call this a *Port Monitor Object*. The approach allows for computational depen-

dent and application specific event generation at the same time, and has the following definite advantages. First, it does not pollute the computational module with application specific details. Depending on the coordination mechanism being used (e.g., state machine, BDI-based system), the required events can be freely generated in the port monitor during application development time. Second, it also simplifies component implementation, since the developer does not necessarily need to be concerned with generating all possible events which can be used in different circumstances. For example, instead of parameterizing Object Detector to generate a coordination event (e.g., “certainty\_low”, “certainty\_high”, “target\_outside\_workspace”), the component can freely output the certainty value (using a separate port or along with 2D position data) which can be used by port monitor to raise the proper events. Moreover, by embedding monitoring into the port, communication and deployment overhead of having a separate monitor component is no longer introduced to the system.

### 5.1.2 Contribution and outline

This chapter proposes some approaches and guidelines to improve the reusability and robustness of robotic systems. More specifically, it alleviates the problem of coordination and reusable component development by embedding data monitoring and arbitration into components’ data flow ports. The contribution of this work can be divided into three parts. Firstly, it proposes the *Port Monitor Object* which extends a component port’s functionality with monitoring and event generation using runtime scripting languages. Secondly, the *Port Arbitrator* and its application to robotics is presented which extends a port’s capability to arbitrate input data from multiple sources. Lastly, it represents some guidelines and further applications of these approaches to improve the reusability of a computational component and simplify its implementation.

The concept of port arbitration and its applications have been discussed in details in Chapter 3. However, one of the limitation of the approach is that the coordination logic can be solely written based on the activation status of the connections. That is, whether they are transferring data or not; and not by considering the data are transmitted through the connection. That limits the system to monitor only the status of connection for coordination and not the data. In this chapter, we introduce a more



sophisticated mechanism where the coordination logic can be written based on the generated events by monitoring the data of each connection.

The rest of this chapter is structured as follows. Section 5.2 describes the port monitor object and its implementation using the YARP framework. The concept of port arbitration, the architecture and its implementation is described in Section 5.3. Further potential applications of the proposed approach are explained in Section 5.4. In Section 5.5 we present the conclusion.

## 5.2 Port Monitor Object

One way to inform a coordinator about state changes of the system is to employ a separate monitor module and configure it with a set of constraints to generate proper events. That is shown in Figure 5.2(a) for coordinating components of the object tracking example from Section 5.1.1. The Monitor component receives data from 3D Position Estimator and generates status events for Coordinator. To increase reusability of the composite subsystem in different architectures or with an alternative coordinator, the Monitor module should offer a generic way to be configured with the required constraints for generating events. Representation of the constraint and logic to raise the events are highly dependent on what is being monitored. Although this can be made to work, it can lead to a suboptimal solution or a nonviable software module. Moreover, the overhead of communication and deployment should also be considered in distributed architectures.

Scripting languages have been used for decades to extend the functionality offered by software components without needing to rebuild or even tweak the base system. These extensions are dynamically loaded and plugged into the component at runtime. Using a plugin system, an alternative approach is to attach a runtime monitor object to the ports of a module. The monitor object is implemented using a scripting programming language and can be loaded by ports at run time.

Figures 5.2(b) and 5.2(c) elaborate the concept of *Port Monitor Object*. As shown in Figure 5.2(b), the port monitor entity (drawn as a box marked M) is attached to the source side of connection between 3D Position Estimator and Arm Control. Using

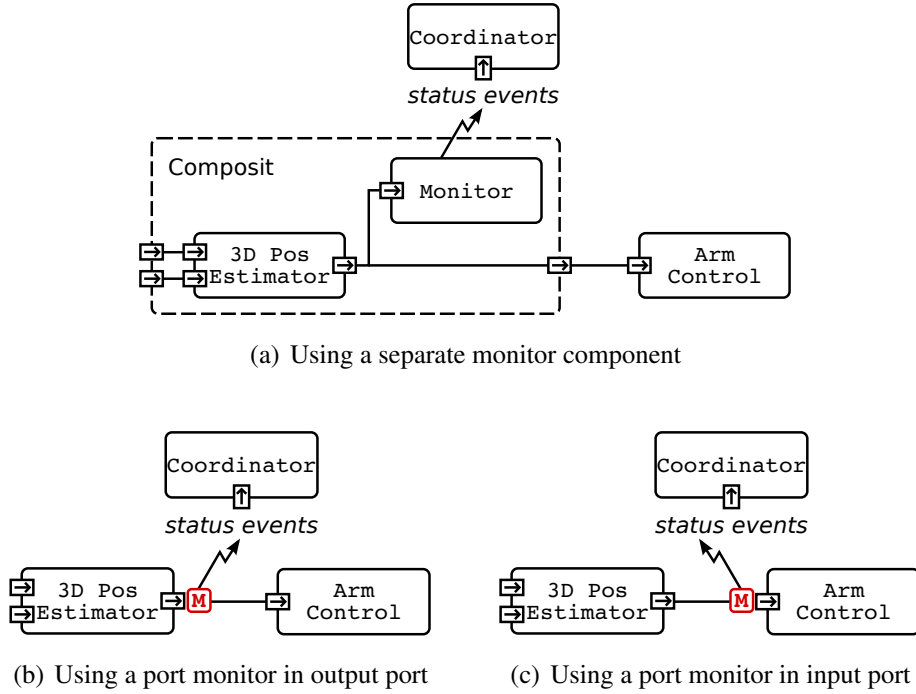


Figure 5.2: Different ways to provide required events for coordinator.

scripting language, users can develop light-weight code to access and monitor the data which is streamed out through the port. Computationally relevant events are now freely generated in the port monitor object in any required format for the coordinator. Alternatively, one can move the monitor object to the other side of connection and attach it to the input port of Arm Control module (Figure 5.2(c)). In this way, the connection between 3D Position Estimator and Arm Control can also be monitored and events can be generated in case of delay or failure in communication.

### 5.2.1 Port monitor life cycle and API

To illustrate the applicability of the described approach, we present the implementation of port monitor objects using the YARP [54] framework. In YARP, programs communicate via units called ports. Messages can be sent between ports, using the connections between them. Connections are not constrained to use the same protocol. A single port may transmit the same message across several connections using sev-

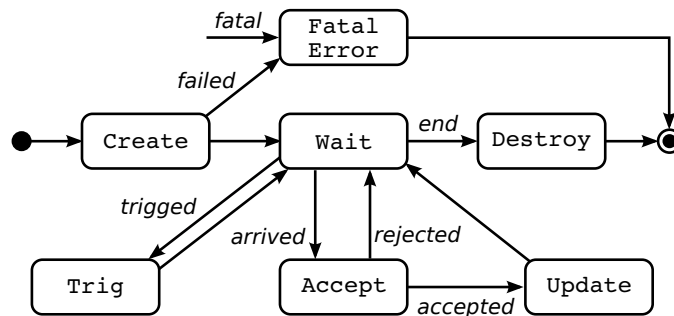


Figure 5.3: The life cycle of port monitor object.

eral different protocols; likewise it may receive messages from several sources using different protocols. The core YARP library does not concern itself with the protocol in use for a connection, except in broad terms (is it reliable? is there a way to include meta-data? are replies possible? etc.) The detailed implementation of individual protocols are encapsulated in plugins called “carriers.” Carriers have a variety of use-cases: running messaging over a new kind of network, allowing successive implementations of protocols to live side-by-side during a graceful deprecation period, supporting network-level interoperation with programs from a different community, etc.

Until recently carriers in YARP have been seen as essentially passive elements, transmitting data in various forms but not actively modifying it. But in fact carriers can be used as hooks that give intimate access to the consumers and producers of data in a network, inserting arbitrary action that is local to a component rather than remote from it. This is the opportunity the port monitor object is building on. A port monitor is implemented\* as a carrier plugin which can be attached to one side of a connection and configured to load a user’s script. For the time being, only the Lua [39] scripting language is supported by the port monitor but it can be easily extended to other languages.

Figure 5.3 illustrates the life cycle of a port monitor object. A callback function is assigned to each state of the monitor’s life cycle (except Waiting) which can have a

\*The source code and relevant examples can be found at [https://github.com/robotology/yarp/tree/master/src/carriers/portmonitor\\_carrier](https://github.com/robotology/yarp/tree/master/src/carriers/portmonitor_carrier)

corresponding implementation in the user's script. Using these callbacks, users have full control over the port's data and can access it, modify it and decide whether to accept the data or discard it. Listing 5.1 represents the callback functions corresponding to the port monitors' states in Lua.

```
PortMonitor.create = function() return true end
PortMonitor.accept = function(rd) return true end
PortMonitor.update = function(rd) return rd end
PortMonitor.trig = function() return end
PortMonitor.destroy = function() end
```

Listing 5.1: Port monitor callback functions in Lua

Monitor's life cycle starts with the Create state where the `PortMonitor.create` callback is called. The initialization of user's code can be done here. Returning a true value means that user's initialization was successful and the monitor object can start watching data from the port. When data arrives to the monitor, `PortMonitor.accept` is called. Using a data reader handler passed to the function, the user can access (for reading only) the data, check it and generate events. The return value of this function indicates whether data should be delivered (accepted) or discarded. If data is accepted, `PortMonitor.update` is called, at which point the user has access to *modify* the data.

A port monitor will usually act as a passive object [59] where `accept` and `update` callbacks are called only upon arriving data. However, one may need to periodically monitor a connection (within a specific time interval) and, for example, generates proper events in case of delay in the communication. For this purpose, a port monitor object can be configured to call `PortMonitor.trig` within desired time intervals. In Section 5.4.1 we demonstrate how `PortMonitor.trig` can be used to monitor the latency in communication. Finally, `PortMonitor.destroy` is called when a port monitor object is detached from the port on disconnection.

Based on the object tracking example from Figure 5.2(b), we show how these callbacks can be used to generate events for the coordinator when the certainty of 3D Position Estimator drops below a desired threshold. Listings 5.2 shows a Lua script which is loaded by the port monitor object attached to the output port of estimator module. In `PortMonitor.create` a YARP port is created which will be used to dispatch events. This allows other modules (e.g., Coordinator) to receive these events by subscribing

to this port. Monitoring data and event generation is done in `PortMonitor.accept`. The port's data is read and the condition for generating the event is checked. If the certainty is below the threshold (e.g., 0.8), `e_certainty_low` is generated and sent out using the dispatcher. Finally, in `PortMonitor.destroy`, the dispatcher port is closed.

```
1  PortMonitor.create = function()
2      dispatcher = yarp.Port()
3      return dispatcher:open("/estimator:event")
4  end
5
6  PortMonitor.accept = function(incoming_data)
7      -- read object_pos from 'incoming_data'
8      if object_pos.certainty < 0.8 then
9          dispatcher:write(event("e_certainty_low"))
10     end
11     return true
12 end
13
14 PortMonitor.destroy = function()
15     dispatcher:close()
16 end
```

Listing 5.2: An example of monitoring data and dispatching events.

## 5.3 Port Arbitrator Object

In robotic applications there are cases where making an immediate decision upon state changes of the system becomes crucial to overall behavior of the complex system [51]. In terms of coordination, it can be much simpler and more efficient to have a reactive decision made quickly rather than introducing delay in the control loop by making every minor (and sometimes inessential) state change of the system explicitly visible to the coordinator.

Figure 5.4(a) shows a simplified architecture of an object tracking application by a robot. Template Matching and Particle Filter modules are configured to recognize a desired object based on different object detection algorithms. The reason for using two

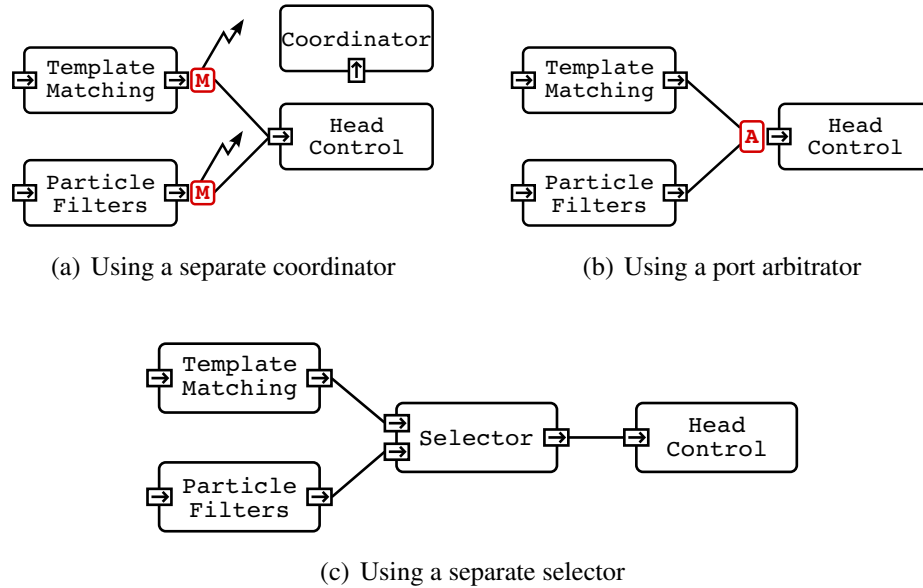


Figure 5.4: Different ways to select desired data from multiple sources.

different modules for the same purpose is that depending on the environmental condition and how object appears in the scene, one algorithm performs better than other. Each module sends 3D position of the detected object along with its certainty to the Head Control component. The latter should receive data from the module which is more confident about its result. This is again a problem of coordination. Yet, how to coordinate these competitive modules? It should be clear that delegating this responsibility to Head Control is not the right choice since it strictly limits reusability of the module.

One solution to this problem is to implement a specific selector which receives data from both detector modules, chooses the one with higher certainty value and sends it to Head Control (Figure 5.4(c)). The drawback of this approach is that, firstly, it introduces overhead of transferring data to and from Selector. Secondly, execution of Selector further delays delivering data to Head Control. Finally, Selector module is hardly likely to be reused in different applications; unless one makes the selector more generic with the cost of lower performance.

Another solution is to monitor output values of each module and make any changes in the certainty level visible to a separate Coordinator using proper events (e.g.,

`e_certainty_low`, `e_certainty_ok`). The Coordinator is then responsible for contacting all the involved modules to inquire each to block or deliver their output to Head Control. This requires extending modules that perform computation and introduce specific logic to enable or suspend sending output data. In case modules are allowed to talk to multiple receivers, this logic should also become aware of the current network topology (for good reasons this information is usually hidden by the middleware). Moreover, proper coordination requires that a certain amount of messages and acknowledgments are exchanged between the modules involved in the arbitration and the Coordinator. This “bureaucracy” introduces latencies, bandwidth overhead and adds complexity to the application.

For this family of coordination problem, we propose another approach based on arbitrating data from multiple sources in the input port of a component. We call this a *Port Arbitrator*. The approach can be used in the design of any robotic system where immediate reaction to changes in the system’s state is required and these minor changes are not necessarily needed to be reasoned about by a third-party component. For the object tracking example, minimizing delays is functionally more important than making every change of the system’s state explicitly visible to the coordinator via events (notice that, although these events are not used by a separate component, they can be made available to higher level decision makers or monitors, if required).

Figure 5.4(b) shows how a port arbitrator object is used in the object tracking example. The port arbitrator (drawn as a box marked A) is attached to the input port of Head Control. The arbitrator is configured with a set of constraints to properly arbitrate between data arriving from Template Matching and Particle Filter modules. As for port monitors, the arbitrator object can be dynamically loaded and plugged into an input port. Thus the communication and deployment overhead of having a separate selector or coordinator component are no longer introduced into the system.

### 5.3.1 Internal Architecture of Port Arbitrator

Figure 5.5 represents the internal architecture of the port arbitrator object. The aim of using a port arbitrator is to allow data from, at most, one connection at a time to be delivered to an input port. A port arbitrator consists of a set of selection constraints,

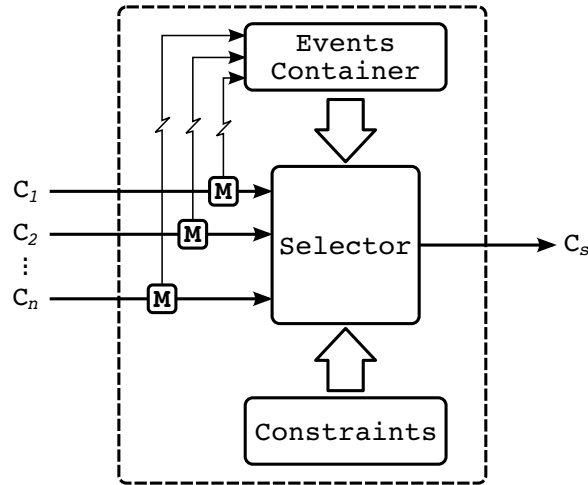


Figure 5.5: The architecture of Port Arbitrator object. Straight lines show the data flow and zigzag lines represent event flow.

an event container and a selector block. In fact, when a port monitor object is attached to an input port, the user's script can access the extended API for arbitration. Listing 5.3 represents the extended port monitor's API in Lua which can be used with port arbitrator.

```
PortMonitor.setEvent(event, life_time)
PortMonitor.unsetEvent(event)
PortMonitor.setConstraint(rule)
```

Listing 5.3: Port monitor extended API in Lua for arbitration

A port monitor object can be attached to each connection ( $C_i$ ) going through the port arbitrator. It monitors the connection and inserts the corresponding events into a shared container. A monitor can also remove an event (if previously inserted by itself) from the container \*. Normally events have infinite life time. This means that they remain valid in the container until they are explicitly removed by the monitor object. An event can also have a specific life time. A time event will be automatically removed from the container when its life time is over. For each connection  $C_i$ , there is a selection constraint written in first order logic as a boolean combination of the names of symbolic

\*This is similar to the Event-Mask mechanism used in user interface programming or in operating systems.



events. Upon the arrival of data from a connection, the selector evaluates the corresponding constraint and if it is satisfied, it allows the data to be delivered to the input port; otherwise the data will be discarded. Clearly a consistency check on the boolean rules must be performed to guarantee that only a single connection  $C_i$  can deliver data at any given time.

### 5.3.2 Representation and Evaluation of Constraints

We refer to the object tracking example from Figure 5.4(b) to demonstrate how selection constraints are represented and how they can be evaluated based on events from a container. As we have mentioned before, Head Control should receive data from the detector module which is more confident about its result. The confidence level is indicated by the certainty value sent out from the module to Head Control. A monitor object is attached to each connection. The monitor reads the certainty value associated with the detected object and inserts an event into the container when the certainty is above a desired threshold. The event is removed from the container if the certainty value drops below the threshold. In our example connections are named  $C_1$  and  $C_2$  for Template Matching and Particle Filter, respectively, whereas the corresponding events are `e_template_ok` and `e_particle_ok`. To allow data from Template Matching ( $C_1$ ) to be delivered to Head Control when `e_template_ok` exists in the event container we add this rule:

```
C1 if e_template_ok
```

A similar constraint should also be set to receive data from Particle Filter. Suppose now we want to give preference to data from Particle Filter if both trackers are confident about their results. This can be achieved by modifying the selection constraint for  $C_1$  (Template Matching) as follow:

```
C1 if e_template_ok and not e_particle_ok
```

As we have described for the object tracking example, constraints can be expressed as boolean combinations of symbolic events. To evaluate the expression, every sym-

bolic event is substituted with a boolean value. If the event exists in the container, it represents a true value in the expression; otherwise it is evaluated as false.

### 5.3.3 Reference Implementation

Port arbitrator is an extended functionality of the port monitor object. It extends monitor's scripting API for setting constraints and altering events in the container. In fact, when a port monitor object is attached to an input port, the user's script can access the extended API for arbitration. To illustrate this, we show how the extended functionality of port monitors can be used for arbitrating connections in the object tracking example of Figure 5.4(b). To do this, a monitor object is attached to each connection of the input port of Head Control. Each monitor object loads a script in which we set the constraints described in section 5.3.2; each monitor alters the corresponding events by parsing the incoming data and evaluating the associated certainty value. Listing 5.4 shows the script for setting selection constraint and monitoring data from Template Matching. The selection constraint (i.e., `e_template_ok` and not `e_particle_ok`) is set in the `create` callback using `PortMonitor.setConstraint`. The certainty value of the detected object is monitored in the `accept` callback. If the certainty is above the desired threshold, `e_template_ok` will be added to the container using `PortMonitor.setEvent`. Similarly, it will be removed from the container using `PortMonitor.unsetEvent` whenever the certainty value drops below the threshold.

```
1 PortMonitor.create = function()  
2     PortMonitor.setConstraint("e_template_ok and  
3                               not e_particle_ok")  
4     return true  
5 end  
6  
7 PortMonitor.accept = function(incoming_data)  
8     -- read object_pos from 'incoming_data'  
9     if object_pos.certainty > 0.8 then  
10         PortMonitor.setEvent("e_template_ok")  
11     else  
12         PortMonitor.unsetEvent("e_template_ok")  
13     end  
14     return true  
15 end
```

Listing 5.4: Setting selection constraint and monitoring data from Template Matching. Setting the selection constraint of Particle Filter and monitoring its data is done in the same way. Notice that each monitor object can set its own selection constraint and only alters its own events in the shared container.

## 5.4 Potential Applications

We have explained how the port monitor object can be used for monitoring data and generating corresponding events. We have also shown how this object is extended to instantaneously arbitrate data from multiple connections. In previous work we demonstrated that this arbitration mechanism can be effectively used to implement complex tasks without resorting to centralized coordinators [66]. In the remainder of this section we show other applications of our approach that can further improve the performance of robotics system and increase component reusability.

### 5.4.1 Monitoring communication for QOS

To achieve robust behavior of a robotic application, the behavior of subsystems and communication among them should be properly monitored. A port monitor object can

be attached to an input port to monitor the connection between the components and raise proper events in case of latency or failure in the communication. The events can be used by a coordinator to control urgent critical situations or monitor the quality of service over longer periods. QOS events can also be used by an arbitrator to select the component which instantaneously provides data with least latency. In the object tracking example from Figure 5.4(b), choosing between data from Particle Filter and Template matching can be done not only based on confidence level of their results, but also by checking which one is producing data with lower latency, higher or just more reliable frequency (i.e. lower jitter). This “quality of service” can vary due to current bandwidth usage in the network connection or computational load of the node in which the module is deployed.

To show how port monitor object can be used for monitoring communication frequency, we refer to the example from Figure 5.2(c) and report the pseudo-code of the script to raise an event whenever receiving data by Arm Control is delayed for a specific time.

```
1  PortMonitor.create = function()
2      PortMonitor.setTrigInterval(0.2)
3      return true
4  end
5
6  PortMonitor.accept = function(incoming_data)
7      received = true
8      return true
9  end
10
11 PortMonitor.trig = function()
12     if received == false then
13         --- raise 'e_qos_not_ok' event
14     else
15         received = false
16     end
17 end
```

Listing 5.5: An example of monitoring communication for QOS.

As shown in Listing 5.5, first we setup a trigger to call `PortMonitor.trig` every 200 *ms*. Whenever data arrives to the monitor object, a flag (`received`) is set. On

every call to `PortMonitor.trig`, the flag is checked and if it has not been set, the `e_qos_not_ok` event is generated (for the sake of brevity, the required code for dispatching events is omitted from the listing). The flag is also reset in the `trig` callback for the next check. In this example, the script only raises an event regarding delay in the communication. Noticed that the check performed in this case is overly simplified but this example can be easily extended in a real application. An interesting extension is monitoring failure in the communication and raising proper events using `PortMonitor.destroy` callback.

### 5.4.2 Data Guarding and Filtering

Developing reusable software is hard; systematically developing high quality reusable software components is even harder [83]. With reusability in mind, there is a risk of *premature generalization* and increased complexity. In other words, to build a reusable component, the developer tries to foresee any future needs and add them as reconfigurable functionalities to the software. Such a commitment may lead to more complex computational components which are polluted with application-dependent functionalities. Imagine that, for the object tracking example from Figure 5.1, we want to limit the operational workspace of the robot's arm to reach for the object only in a specific region. One way to achieve this is to configure the 3D Position Estimator module to send object position if it is within the desired region. The problem is that the output of the estimator module is also used by Head Control. Thus, it also limits the operational space of the robot's head. Another approach is to delegate this responsibility to the Arm Control module by configuring it to accept the position data if it is within the desired limit. However, if the Arm Control is mutually used by other modules (which need to control the arm in different workspaces), it should be reconfigured every time it receives data from a different module.

A more flexible approach is to use a port monitor object (e.g., in the output port of 3D Position Estimator) and constrain it to pass the data within the desired limit. That can be done by monitoring data packets in the `PortMonitor.accept` callback and rejecting any that do not satisfy application's needs. In this way, developers are not forced to include any application-dependent functionalities into their components. Hence,

components can be reconfigured with the parameters which purely affect their computational functionalities.

### 5.4.3 Data Transformation

Components exchange data through their ports. To establish a meaningful communication, they should commonly agree on the type of exchanging information. Based on data type, Brugali et al. [15] classify them into strongly-typed and loosely-typed and discuss the pros and cons of each category. Strongly-typed communication is known to be more efficient and easier to debug but at the same time limits the reusability of component. In contrast, loosely-typed communication is more flexible but requires more manual programming; that is the interpretation of messages should be handled in the component implementation.

Scripting languages due to their text-processing capabilities have been known to be well suited to the task of data transformation and munging [23]. Port monitor approach allows for data modification (`PortMonitor.update`) using scripting programming languages. Thus, it is potentially an ideal place for basic data conversion. One can attach a port monitor object to an input port of a component and implement a simple script to take the data and convert it into the format which is required by the component. Moreover, using port monitor for loosely-typed data mapping, we simplify component implementation since the component does not need to bear the responsibility of interpreting information.

### 5.4.4 Logging and Performance Monitoring

To analyze the runtime performance of a robotics system, the behavior of components, their interactions and in general, any critical state changes in the system should be monitored over long periods. This is analogous to the Top-Down passive monitoring in the field of application performance management [25]. Passive monitoring is usually an appliance which leverages network port mirroring. The idea of port monitoring can be applied to record the quality of service provided by a computational component over time. The only way components can communicate with the external world is via

their ports. Through explaining port monitor object (Section 5.2) and its application (Section 5.4.1), we have clearly shown how port monitor object can be used to generate both computational and communicational events. Thus, these events can be locally recorded by port monitor for off-line analysis or easily sent to a central events logger.

## 5.5 Conclusions

This chapter has introduced port's data monitoring and arbitration to alleviate the problem of coordination and facilitate development of reusable components. We have illustrated the port monitor object and how it extends a component port's functionality with monitoring and event generation using runtime scripting languages. To demonstrate our approach and its potentials we presented a reference implementation of each approach using the YARP framework.

We showed that our approach allows separating the computation from application dependent code. This increases the reusability of the components and it simplifies their implementation. We also demonstrated that how port monitor object can be used to implement data filtering and transformation, and implementing quality of service and performance monitoring. Overall this can substantially improve the robustness of robotics application.

We proposed an extension of port arbitration and how we enhanced the port's capability to arbitrate input data from multiple sources based on the rules written using generated events. Our approach to port arbitration can also contribute to improving the performance of a robotic system when changes to the system's state can be kept local to certain components and immediate reaction is required. However, choosing between an explicit coordination mechanism and more reactive but less explicit way of orchestrating components, is a design choice and depends on functional requirements of robotics system.

# Chapter 6

## Enhancing software module reusability using port plug-ins: an experiment with the iCub robot\*

### 6.1 Introduction

Robotics software community is continuing to grow. Within the community, researchers have been developing large number of software components using some of the most common robotic middleware, such as ROS [71], YARP [35], OROCOS [17], OPROS [42] and Open-RTM [5] or based on their customized frameworks using standard communication libraries (e.g., CORBA [63], ICE [41], ØMQ [3]). They try to adopt lessons learned from best practices in robotics [15, 16] and software architecture techniques and standards [74] to build their modules as reusable as possible. Even so, it is quite unlikely that components from different communities fit into a specific off-the-shelf deployment scenario, without any adaptation by third party users. Heterogeneity and lacking standards are not the only bottlenecks burdening reusability. Even within a community of developers who share the same middleware, software

---

\*This chapter is based on Ali Paikan, Vadim Tikhonoff, Giorgio Metta and Lorenzo Natale. Enhancing software module reusability using port plug-ins: an experiment with the iCub robot. *International Conference on Intelligent Robots and Systems - 2014* (submitted).



components can be developed with different taste and still hard to reuse. Systematically developing high-quality reusable software component is, indeed, a difficult task. Many developers keep their modules simple. However, simplicity does not necessarily lead to more reusable software. On the other hand, with reusability in mind, there is a risk of over-generalization and increased complexity: to build a more generic and reusable component, the developer tries to foresee all possible future needs and add them as reconfigurable functionalities to the software. Such a commitment leads to complex components, polluted with application-dependent functionalities that are more costly and difficult to maintain and use correctly. Thus, a proper balance must be found between potential reuse and ease of implementation [74].

Software should be extensible enough to be adapted to possibly unanticipated changes [87]. Extensibility is an important property for software which significantly boosts reusability. One direction to extend a module is via its interfaces. In distributed systems interfaces are implemented by exchanging messages through special connection points that are call ports. This plays an important role in nowadays robotic software architectures. In Chapter 5 we have explained how the port's functionalities is extended in order to dynamically load a run-time script and plug it into the port of an existing module without changing the code or recompiling it.

Plug-in platforms, in general, extend a core system with new features implemented as components that are plugged into the core at run time and integrate seamlessly with it. When an application supports plug-ins, it enables customization, thus, provides a promising approach for building software systems which are extensible and customizable to the particular needs [85]. Probably one of the more prominent example of a platform which broadly supports plug-ins is Eclipse IDE [30]. Eclipse offers a framework to develop plug-ins in Java which are delivered as JAR libraries. There are also some generic frameworks for plug-in development and management such as Pluma [2] which allows loading plug-ins as dynamic linked libraries or FxEngine [1] for data flow processing and the design of dynamic systems. plug-ins can also be developed using scripting languages. Scripting languages have been used for decades to extend the functionality offered by software components and they have special interests within the game developer communities. The main advantage of script-based plug-ins is that they are usually easier to be developed and maintained. Despite plug-in system has

been broadly used by software developers over the last decades, to our knowledge, less attention has been devoted to study their potentials in the robotic field.

This chapter presents a practical application of port plug-ins using iCub humanoid robot [53] and demonstrates its distinct advantages in software reuse. The experiment is completely built using modules from the iCub software repository\*. It focuses on reusing (with no modifications) existing modules by extending the required functionalities using port plug-ins.

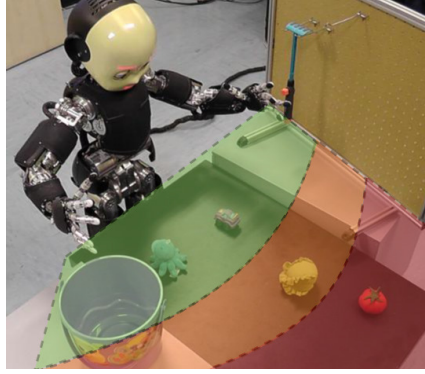
## 6.2 A step-by-step example

The overall behavior of the experimental task is demonstrated using a simplified activity diagram in Figure 6.1(b). The goal of the task, as shown in the activity diagram, is to clean the table by removing all the object and place them in a bucket located alongside the table. We allowed the robot to use a tool at his disposal (a rake), located on a rack, to reach objects of interest that are out of his workspace. The modules that allow the robot to grasp and use the tool are implemented as described in [79]. Furthermore, we consider also the case in which the object is so far that it cannot be reached even by the use of the tool. In this case the robot should look for a human and asks his intervention (put the object within reach). Figure 6.1(a) shows the experimental setup and it illustrates the three areas in which objects can be placed.

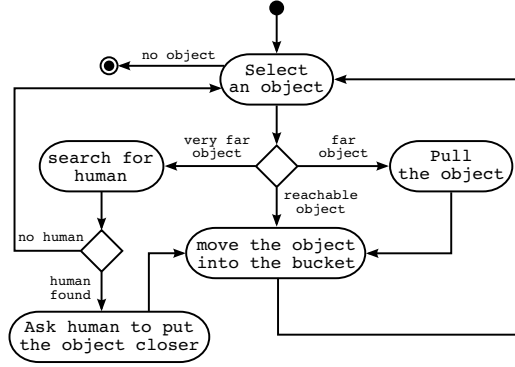
The activity diagram depicted in Figure 6.1(b) may give the impression that the task is only composed of a few simple steps that the robot should follow to accomplish it. But in fact, there are many uncertainties and unexpected conditions which should be taken into consideration to make the task robust. For example, the proper decision should be taken if an object drops from the hand while the robot is placing it into the bucket. Similarly the robot should behave appropriately while it is holding the tool to pull the object closer, the human might intentionally intervene and move the object within the iCub's workspace. Considering all possible uncertainties, in fact, reveals the underlying complexity of the task which requires that many modules (e.g, for perception,

---

\*Modules can be downloaded from: <https://github.com/robotology/icub-main.git> and <https://svn.code.sf.net/p/robotcub/code/trunk/iCub/contrib>



(a) The experimental setup.



(b) The simplified activity diagram.

Figure 6.1: The experimental setup and the simplified activity diagram of table-cleaning application. The reachable zone is depicted in green, the orange zone represents the zone reachable with the tool and finally the red zone indicates the unreachable space, for which the robot needs human intervention.

action and coordination) are properly used and orchestrated (e.g, coordinating robots, gaze, arm, speech) to perform the required task.

Table 6.1: A subset of modules used for the experiment

Module	Input	Output	Type
<b>Face-Detector</b>	image	pos_3D	perception
<b>Object-Detector</b>	image	List<pos_3D>	perception
<b>Bucket-Detector</b>	image	pos_3D	perception
<b>Look-Around</b>	-	pos_3D	implicit action
<b>Head-Control</b>	pos_3D	-	action
<b>Pick-and-Place</b>	msg_cmd	msg_status	action
<b>Pull-Object</b>	msg_cmd	msg_status	action
<b>Speak</b>	msg_text	-	action

The modules used in this experiment are chosen from the iCub software repository and listed here in Table 6.1. To build the desired application, a few modules might simultaneously require to grab the camera image frames from the robot, control the arms and hands in various modes, such as Cartesian or joint space using velocity or

position control. However, for the sake of brevity, only a subset of these modules are described here. We use the previously mentioned Face-Detector and the Look-Around modules.

Object-Detector gets as an input image from the cameras and produces a list of blobs and extracts 3D positions of all the possible graspable objects as its output. Bucket-Detector is, in fact, an instance of a generic object detector which is configured and trained to recognize this specific object. As we previously mentioned, Look-Around randomly produces positions in 3D space which are used by Head-Control to move the gaze in various positions. The Pick-and-Place module receives a set of commands (e.g., take <3D\_pos>, put <3D\_pos>) to take an object and release it on a specific position. The internal status of the module (e.g., e\_taken, e\_arm\_idle) is continuously sent out using status messages. Pull-Object is a complex set of modules which together get the position of an object on the table and use a tool to bring the object closer [79]. Similar to Pick-and-Place, the internal status of the Pull-Object module is advertised via its output. The Speak module receives a text message and performs a text-to-speech synthesis. Generally speaking, in order to be able to integrate some modules for building an application, two important points should be considered: *i*) data type on both side of the connections should match and *ii*) a proper coordination mechanism should orchestrate modules to perform the task. We start with the simplest case in which the objects are reachable by the iCub and progressively extend it to build the complete table-cleaning application.

### 6.2.1 Handling reachable objects

First our application should select the closest object within the reachable area and take it (see Figure 6.2-A ). To do that, we connect the output of Object-Detector to the input of Pick-and-Place. Using the port monitor, we implement a simple script that goes through the list of objects, select the one that is closest to the robot and produces the proper ‘take’ command (i.e., take <3D\_pos>) for execution. Similarly, to put the object into the bucket we connect the output of Bucket-Detector with the same input of Pick-and-Place and attach to this connection another port monitor that generates the ‘put’ command (i.e., put <3D\_pos>) for execution (see Figure 6.2-B )

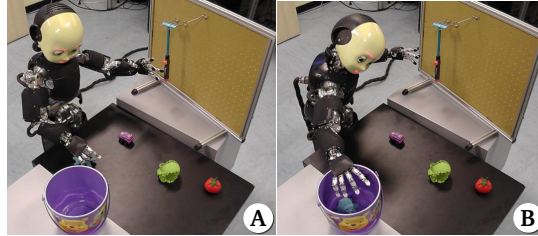


Figure 6.2: The iCub performing table-cleaning on reachable objects. The robot takes the object (A) and places it into to bucket (B).

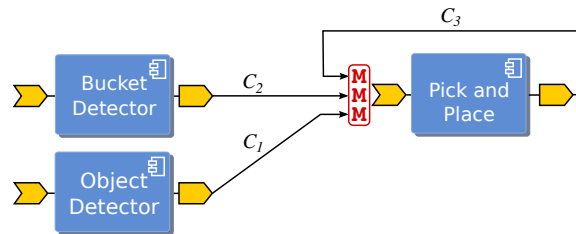


Figure 6.3: Configuration of the modules for handling reachable objects on the table.

Furthermore, an object should be taken only if the hand of the robot is free and the robot is not performing another action using the arm. On the other hand, the ‘put’ command should be sent to the Pick-and-Place module if the robot is holding an object. To this aim, the status of the Pick-and-Place module should be monitored and the required arbitration rules should be added to the system to properly coordinate taking, placing and releasing actions. Figure 6.3 represents the configuration of the modules that perform this simple task on the reachable objects. As shown in the figure, the status output of the Pick-and-Place module is used to inform the arbitrator about the internal state of the module. Below we illustrate how this is achieved.

As we have previously mentioned, a monitor object is assigned to each connection going through the port arbitrator. Listings 6.1, 6.2 and 6.3 respectively represent pseudo-scripts which will be loaded by each monitor object for connections  $C_1$ ,  $C_2$ , and  $C_3$ . Listing 6.3 demonstrates the script which is assigned to the monitor object of connection  $C_3$ . This monitor receives status messages from Pick-and-Place (i.e., `e_taken`, `e_arm_idle`) and adds them to the event container of the port arbitrator. These events

will be used for the selection of  $C_1$  and  $C_2$ . Notice that the connection  $C_3$  and the corresponding script (Listing 6.3) are created to make the status events available for the arbitration. These events will be never delivered to Pick-and-Place. This is achieved by refusing to accept the data from the connection  $C_3$  (return false).

```

1  PortMonitor.create = function()
2      setConstraint("not e_taken and e_arm_idle")
3      return true;
4  end
5
6  PortMonitor.accept = function(object_list)
7      -- find closest_obj in the object_list
8      if closest_obj.dist > HAND_REACHABLE then
9          return false
10     end
11     return true
12 end
13
14 PortMonitor.update = function(object_list)
15     return command("take", closest_obj.pos)
16 end

```

Listing 6.1: Monitoring and arbitrating connection  $C_1$ .

```

1  PortMonitor.create = function()
2      setConstraint("e_taken and e_arm_idle")
3      return true;
4  end
5
6  PortMonitor.update = function(bucket_pos)
7      return command("put", bucket_pos)
8  end

```

Listing 6.2: Monitoring and arbitrating connection  $C_2$ .

```

1  PortMonitor.accept = function(status_event)
2      setEvent(status_event, 0.5)
3      return false
4  end

```

Listing 6.3: Monitoring connection  $C_3$  for generating events.

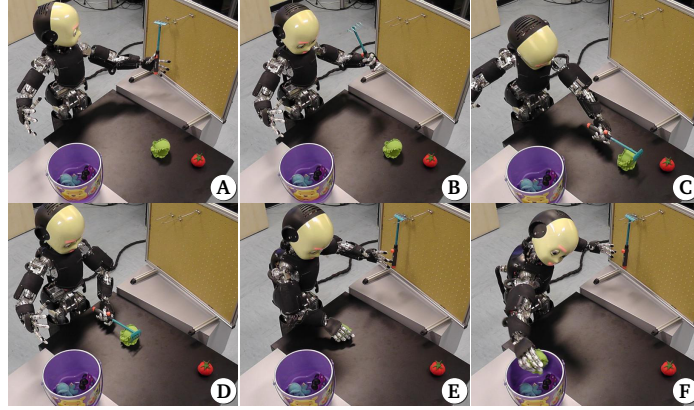


Figure 6.4: The iCub performing table-cleaning using a tool (rake). The robot take the tool (A), reaches for the object (B,C), pulls the object (D), grasps the object (E) and finally places it into the bucket (F).

Listing 6.1 deserves particular attention: First, within the ‘create’ callback, the required selection rule for the connection  $C_1$  is set into arbitrator. The rule implies that data from corresponding connection should be delivered if the robot has not already taken (not `e_taken`) an object and if it is not performing an action (`e_arm_idle`). In the ‘accept’ callback, first the closest object to the robot is selected from the list of detected objects. If the object is reachable (the data is accepted), the ‘update’ method will be called to generate the ‘take’ message to be delivered to Pick-and-Place. If the object is out of reach, it will be discarded (`return false`). Similar Listing 6.2 represents the script that generates the ‘put’ command and that specifies the condition under which performing the corresponding action.

### 6.2.2 Handling objects using tool

We now extend the previous application to allow the iCub to use a tool to bring unreachable object within its workspace (see Figure 6.4 ). Figure 6.5 represents how Pull-Object is integrated in the application. The output of Object-Detector module provides a list of objects; this list should be filtered to select one object that is within the tool-reach area and out of the robot’s workspace. The position of this object should be given to the Pull-Object to trigger a sequence of actions to take the tool from the

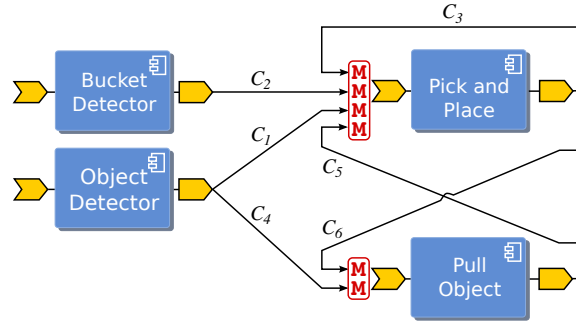


Figure 6.5: Configuration of the modules for handling objects within tool-reach space.

rack, reach for the object with the tool, pull the object and finally putting back the tool on the rack (see Figure 6.4-B, C, D ).

```

1  PortMonitor.create = function()
2      setConstraint("not e_taken and e_arm_idle")
3      return true;
4  end
5
6  PortMonitor.accept = function(object_list)
7      -- find closest_obj in the object_list
8      if closest_obj.dist > TOOL_REACHABLE then
9          return false
10     end
11     return true
12 end
13
14 PortMonitor.update = function(object_list)
15     if closest_obj.dist < HAND_REACHABLE then
16         return command("cancel", nil)
17     end
18     return command("pull", closest_obj.pos)
19 end

```

Listing 6.4: Monitoring and arbitrating connection C<sub>4</sub>.

Once the object is located within the reachable area of the robot, the previous picking-and-placing application is activated. Appropriate selection rules should be added to the system to properly arbitrate pulling and pick-and-placing.



Listing 6.4 represents the pseudo code of the script which is used in the port monitor of connection  $C_4$ . The selection constraint (`not e_taken and e_arm_idle`) filters messages to Pull-Object when the robot is already involved in other actions (i.e. picking and placing an object). Similar to Listing 6.1 from the previous application, first the closest object is extracted from the list of detected objects. This object is accepted and generates a ‘pull’ command if it is within the tool-reach area. Otherwise it is discarded. An interesting behavior is the fact that the pulling action is composed of several sub-actions that should be aborted if the tool becomes unnecessary (e.g. if a human moves the target objects in the workspace of the robot). This is achieved by continuously monitoring the target object in the ‘update’ function and generating the ‘cancel’ command when necessary. Notice that as opposed to Pick-and-Place, Pull-Object ignores redundant ‘pull’ commands until all ongoing sub-actions are accomplished or aborted (with the ‘cancel’ command). Therefore, unlike Pick-and-Place, we do not need to monitor the internal status of Pull-Object and filter conflicting ‘pull’ commands.

Clearly Pick-and-Place and Pull-Object are conflicting behaviors. To avoid conflicts the selection rule for connection  $C_1$  must be updated to prevent generation of ‘take’ commands while Pull-Object is active (i.e. not idle). This is achieved by making the internal state of the Pull-Object available in the arbitrator of Pick-and-Place via connection  $C_5$  and by modifying the selection constraint of Listing 6.1 as follows: ‘`not e_taken and e_arm_idle and e_pull_idle`’

As for the connection  $C_3$ , Listing 6.3 is used for the port monitor of connections  $C_5$  and  $C_6$  to insert the status events into the corresponding event containers.

### 6.2.3 Handling objects with human assistance

In Chapter 4 we explained how the Face-Detector and Look-Around modules can be properly used with the Gaze Control (i.e., similar to the Head-Control module) to implement a basic face tracking application. In this section, we use these modules to complete our table-cleaning application. When an object is completely unreachable, the robot should look for a person and asks assistance (see Figure 6.6). Figure 6.7 depicts the complete system. The output of Object-Detector arbitrates the connections from Face-Detector and Look-Around via  $C_7$  and  $C_{11}$  so that when

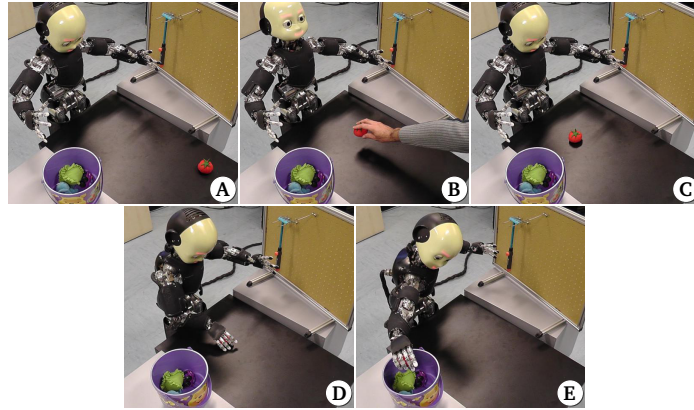


Figure 6.6: The iCub performing table-cleaning with human assistance. The robot detects an unreachable object (A), detects the presence of a human and asks assistance (B,C), grasp the object (D) and finally places it into the bucket (E).

required, the robot will look around searching and tracking human faces. This is achieved in Listing 6.5 by monitoring the closest object and generating an event 'e\_unreachable' when the latter is out of the tool-reach area. Notice that this event is cleared (removed from the container) only when the object becomes reachable again.

```

1  PortMonitor.accept = function(object_list)
2    -- find closest_obj in the object_list
3    if closest_obj.dist > TOOL_REACHABLE then
4      setEvent("e_unreachable")
5    else
6      unsetEvent("e_unreachable")
7    end
8    return false
9  end

```

Listing 6.5: Monitoring connection  $C_7$  and  $C_{11}$ .

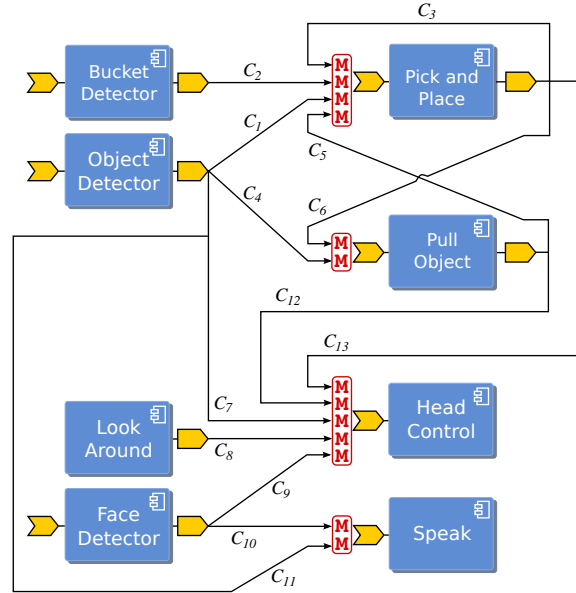


Figure 6.7: Configuration of the modules for table-cleaning application.

```

1  PortMonitor.create = function()
2      setConstraint("e_unreachable")
3      return true;
4  end
5
6  PortMonitor.accept = function(data)
7      if time() - time_prev < DESIRED_TIME then
8          return false
9      end
10     time_prev = time()
11     return true
12 end
13
14 PortMonitor.update = function(data)
15     return msg("Please put the object closer!")
16 end

```

Listing 6.6: Monitoring and arbitrating connection  $C_{10}$ .

Messages from Look-Around and Face-Detector are discarded depending on the internal state of Pick-and-Place and Pull-Object via connections  $C_{12}$  and  $C_{13}$  and the event

generator script (i.e., Listings 6.3). This prevents moving the head when the robot is picking, placing or attempting to pull an object. Finally the output of Face-Detector generates a voice message synthesized by the Speak module. This is achieved by connecting the two modules ( $C_{10}$ ) and adding a script to the corresponding port monitor. This script generates a text message (a valid command for the Speak module) if a human face is detected, but only if a certain amount of time has passed from the last command, to reduce verbosity (Listing 6.6). Notice that these commands are arbitrated by  $C_{11}$  so that the speech is activated only when necessary.

## 6.3 Conclusions

To demonstrate the potential advantages of our approach, we illustrated the design and implementation of a complex application on the iCub humanoid robot. The application was completely built out of existing modules without code changes. All the functionality specific to the application were implemented and integrated as plug-ins scripts.

The key idea of our plug-in system is to extend modules functionalities by adding scripts to the ports that allow data monitoring, filtering, transforming and arbitration. The main advantage of our approach is that it allows to limit application specific functionalities to scripts that are external to the modules and are added and executed at runtime. This maintains modules clean from unnecessary complexity and enhances their reusability. Finally, by using embedded scripts inside the ports, we can avoid introducing specific modules to achieve the required functionalities, thus, reducing communication and deployment overhead.

# **Chapter 7**

## **Application description and management model in YARP**

### **7.1 Introduction**

Autonomous robots have evolved into complex systems which require many concurrent activities to collaborate and interact in a distributed environment. Designing control system with multiple processes running on a set of machines seems to be a good compromise between performance and the time spent struggling for code optimization [54, 70]. Code reuse and real-time constraints motivate researchers even further toward using distributed modular frameworks [5]. A similar trend can be seen in the automation and industrial robotic systems where smart sensors and actuators indicate remarkable benefits in having more computational power of distributed systems [67].

Despite many decades of work in robotic, we still lack a common robotic middleware. Research objectives, operational requirements, uncommon consensus and many other issues prevented the emergence of a well-accepted framework [77]. Robotic developers from different research laboratories have applied software engineering tools in designing and integrating the modules that compose modern robotics apparatus [34]. Thus, many robotic frameworks which share similar design principles, have evolved independently [17, 20, 40, 54, 71]. The key concept of these frameworks is support for modular or component-oriented architectures where software modules can be dis-

tributed over a cluster of machines and data is shared via well-defined interfaces or connection ports. Peer-to-peer architectures (e.g. publish/subscribe) [26] are widely used in this respect. Although connections can be established automatically inside the modules, it is far preferable to factor out details concerning interconnections between modules. The advantage of this is that connections can be established dynamically and differently depending of the execution context, thus favoring reusability and portability. It is therefore more common to interconnect modules at run-time or when specific applications are designed.

Much work has been devoted to provide users with useful APIs for module implementation, integration and communication; unfortunately scant attention was devoted to module management on a cluster of computers. Scalability, dependency and portability of applications in cluster of machines are mostly left in the background by robotic researchers. We believe this has strong implications on the field since it prevents research to scale up from tackling relatively simple scenarios that focus on specific problems addressable with few modules, to complex scenarios that require proper execution and cooperation of several modules working together.

This chapter introduces a systematic method to enhance scalability and management of software modules in distributed robotic architectures. We specifically focused on practical difficulties that, as developers, we encountered in dealing with execution and monitoring of large number of modules on a cluster of machines. The work we propose is the result of our long-run experience in developing humanoids robotic applications using the YARP framework [54].

The remainder of this chapter is organized as follows. In Section 7.2 we motivate our work and review relevant approaches in the literature. Section 7.3 discusses the main concepts which are used to model the problem. In Section 7.4 we describe the representation we use and an algorithm for reasoning on the representation (i.e. solving dependencies and connections between modules). Section 7.5 illustrate an example of the implementation of our approach in YARP which also demonstrates how it can be adopted by a robotic framework. Finally in Section 7.6 we discuss the conclusions.

## 7.2 Motivation

In distributed programming paradigm a complex control system is broken down into simpler tasks known as components or modules. The latter are implemented as executables or dynamic linked libraries which can be executed using a specific deployer. Each module has a set of input ports to receive data and deliver the results of the computation to its output ports. Alternatively, in some architectures, input and output might be implemented as a unique bidirectional interface. Modules can be distributed over a cluster of computers and based on the communication architecture and protocol (e.g. standard TCP/IP, CORBA, ICE), the outputs of one module can be delivered to one or many inputs of other modules.

Using the above-mentioned paradigm, developing robotic applications involves two stages: 1) implementing modules and 2) testing, i.e. running and establish connections between modules. The second stage is particularly important in research environment, because modules are switched in and out to test performance, experiment with alternative algorithms, different hardware and configurations. This is indeed the main concern of our discussion, which is discussed throughly in this section.

### 7.2.1 Dependencies

In a collaborative environment, many modules are executed together and information are properly transferred from one to another, literally forming a hierarchical data follow. This, in fact, implies a hierarchical dependency among modules. While developing an application, users must be aware of these dependencies, understanding module's inputs and outputs, data type and which outputs can be candidates for an input. Some of the robotic frameworks [5] provide specific utilities for developing applications but automatic dependency resolving are not well addressed and their approaches are usually tighten to their architecture.

### 7.2.2 Interconnection

Developers should have clear understanding of each module's interface and the properties which are needed to be properly set for interconnecting modules. Employing modules in behavior-based architectures or state-machines might require some extra connections to be made for controlling their activities. As a consequence execution of large applications brings overhead associate to the task of managing connections and connection parameter; these tasks are usually error prone and tedious if done manually but can easily be automated.

### 7.2.3 Integration and composition

Robotic applications in complex scenarios can suffer of limited scalability as the number of modules and connections increase. Having different levels of application granularity in which modules can be integrated and grouped as sub-applications, seems to be reasonable approach. However, there are some issues which should be dealt with when different instances of an application are employed. [82] proposes a solution for plugging modules into groups and propagating data through different levels. Although this looks like a satisfactory solution, it still requires that specific features are implemented and made available in the underlying software middleware.

### 7.2.4 Execution and monitoring

Eventually modules should run on a machine or remote machines. Remote deployment and execution of modules, in fact, is not a new issue and has been addressed by many researchers from different fields. In the field of cluster and grid computing, there are verity of approaches and tools for application description [46], software deployment [19, 27] and monitoring [28]. Some frameworks have their own deployment protocol (e.g. yarprun from YARP) or other tools based on some standard approaches (e.g. `deployer-corba` from OROCOS, `roslaunch` from ROS). What we are interested in here is not actually how to remotely launch a program. Instead, we look at how to use the available approaches in a systematic way to monitor modules and execute them with respect to their dependencies. For example, execution of a module might



need to be delayed until another activity has been performed or some interfaces are initialized by other modules. Moreover in our approach, we require multi-platform (e.g. Windows, Linux) execution and automatic interconnection of modules.

### 7.2.5 Application migration and load balancing

Robotic systems are heterogeneous and robotic applications are highly platform dependent [77]. Due to intensive hardware dependencies, developers mostly prefer to manually decide which module should run on which machine. This, in the one hand, solves the dependency problem but on the other hand hinders application migration and load balancing of modules in cluster of computers. In [11] a generic model and an architecture for a dependable deployment is presented. Similar approaches with slightly different modeling can be also seen in [19, 27]. Although these approaches propose strong logic for reasoning over dependencies, they do not strongly support intensive device-dependent applications. Load balancing is also widely addressed in the field of grid computing [56]. However, many of load-balancing approaches should be employed and integrated with dependable deployment methods to fulfill robotic applications requirements.

### 7.2.6 Cross-middleware deployment

As stated in the introduction several software middlewares have been developed in the robotics community. Reusability of software modules from other frameworks is therefore an interesting topic. The bottleneck mostly resides on the communication interface in which some frameworks provide specific solution to interconnect with others (e.g. YARP offers tools and protocol to communicate with ROS). Apart from interconnection, a variegated application which consists of modules from different middlewares, requires a generic multi-framework deployment tool. We do not claim our approach effectively solves this problem. In fact, we do not believe that such a deployment tool exists. However in Section 7.4 we provide a deeper insight into this issue.

## 7.3 Conceptual Representation

In this section we provide a conceptual representation of the elements which define the semantic of our work. The concepts can be expressed using any classic conceptual modeling approaches or modern ontology representation. They are used to formalize the problem and the requirements that have been discussed in the previous section. For sake of brevity, some of details and properties are not shown in corresponding figures.

### 7.3.1 Resource

A resource refers to any physical or logical devices which are available in a cluster of machines or required by a program. As shown in Figure 7.1, a *GenericResource* is categorized as *PhysicalResource*, *LogicalResource* and *CompoundResource*. Every hardware devices can be seen as *PhysicalResource*. Operating systems, libraries, device drivers and etc belong to *LogicalResource*. A *Computer* is a *CompoundResource* which has a set of peripheral resources, logical resources and all of the primary resources. *PrimaryResources* are processors, main storage, memory and standard network device. Properties value of primary resources (e.g. the architecture, number of cores, load average of a *Processor*) are automatically recognized by the system resource discoverer.

Each computational node of a cluster can be expressed as a *Computer*. Any extra resources (e.g. GPU, shared library), can be specifically expressed by their properties and values as either *PeripheralResource* or *LogicalResource* of *Computer*.

### 7.3.2 Module

A module is a separate, interchangeable component which accomplishes a task or activity and contains everything necessary to accomplish this. Module is typically incorporated into the program through its *Interfaces*. In other words, it receives *Data* with specific *DataType* from *Input* and send the results through *Output*. Some input data are

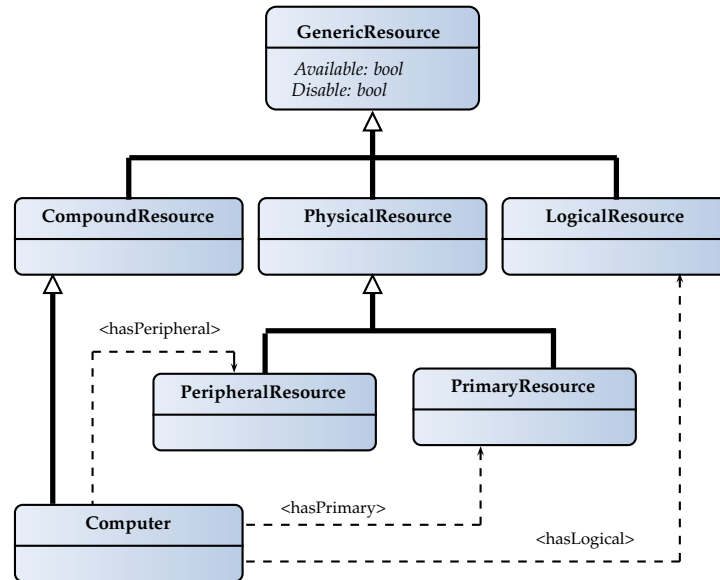


Figure 7.1: Conceptual representation of resource.

required for a module to be able to operate properly\* and some should be available for initialization prior to execution of the module. In Figure 7.2 these are shown as required and priority properties of *Module*. *ModuleProperty* and *InterfaceProperty* respectively are framework-dependent properties of module (e.g. command-line parameters, environmental variables) and interface (e.g. connection type). Rank indicates the popularity of a module among the users with respect to the other modules which accomplishes similar activity. It can be given based on module's computational cost, quality of services it provides and etc.

A *Module* might depends on specific hardware devices, libraries, platform and etc. In our conceptual model it is shown as resources of *Computer*. Every module's resource dependencies can be expressed within multiple *Computer* concepts which are combined in form of disjunction and conjunction. Thus, for example, one can say a module requires a computer with GPU **or** a computer with at least four processor's cores **and** more than one gigabyte free memory space. In Section 7.5 we will show how it can be represented using XML language.

\*Notice that not all of the inputs are required by a module such as those that involve requesting status or specific services.

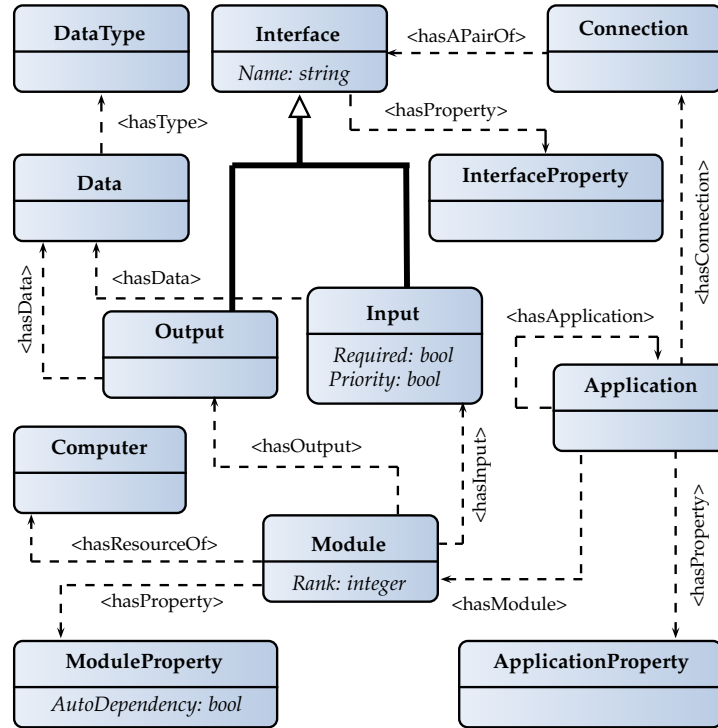


Figure 7.2: Conceptual representation of module and application.

### 7.3.3 Application

An application consists of collections of *Modules* which are interconnected by a set of *Connections* and collaborate to accomplish an specific goal. For example, a grasping application can be made of attention system, gaze control, reaching and grasping modules. A *Connection* is composed of a pair of source–destination interfaces with their properties. To achieve decreasing levels of granularity, different applications can be integrated as sub–applications (for example the grasping application described above can be used as a building block in a larger application in which the robot has to fetch objects from the fridge and bring them to the user). Some global properties of modules or connections can be introduced using *ApplicationProperty*. It can be advantageous when different instances of one application are integrated.

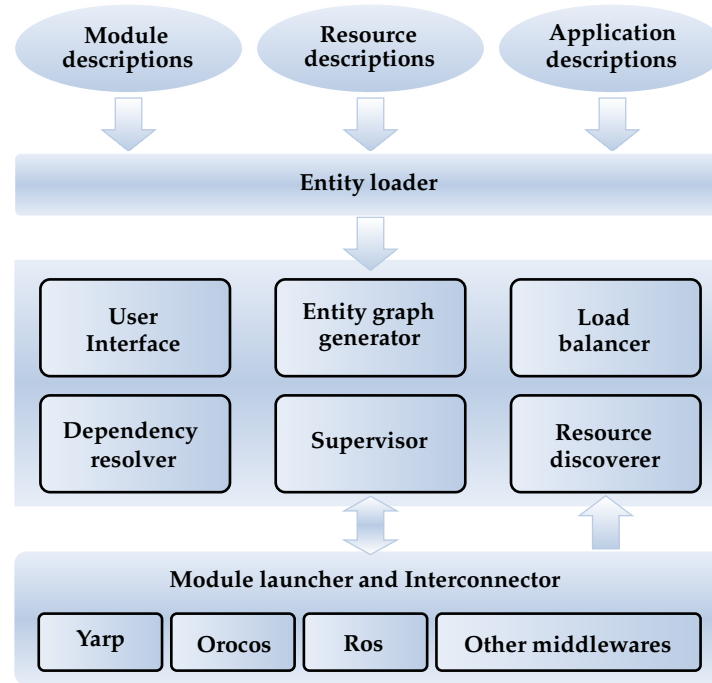


Figure 7.3: Multilayer architectural design of Yarpmanager.

## 7.4 System Architecture

In an architectural perspective, the system is composed of different components, which are combined in a multilayer architecture as shown in Figure 7.3. Three separate layers can be identified: Entity layer on the top, System core and manager in the middle, and an abstract Module Launcher and Interconnector for module execution and interconnection at the bottom.

For each implemented module in a framework, there should be a Module description (e.g. in form of metadata or manifest file) which recites the module's characteristics based on the conceptual model explained in the previous section. Resource description details available machines in a cluster according to the *Computer* conceptual model. In the simplest form, it is a name-list of cluster's machines (e.g. hostname or IP address). Similarly, application's properties, integrated modules and their connections are expressed using Application description. Entities

which are listed in module, resource and application descriptions are loaded by Entity loader and structured in a dependency graph using Entity graph generator. Dependency resolver reasons on the graph to find a set of modules to be executed and the relevant connections to be established. Resource discoverer is responsible for discovering and updating *PrimaryResources* of clustered machines (e.g. CPU's load average, free memory space) which are used by Load balancer to assign each module to the proper machine. Eventually, modules are robustly launched, interconnected and monitored by Supervisor using appropriate middleware-dependent launcher and interconnector from deployment layer. This layer can be enhanced with multiple deployers from different frameworks for cross-middleware deployment.

#### 7.4.1 Dependency graph

The graph generated by Entity graph generator deserves particular attention. Figure 7.4 represents an example of dependency graph which is automatically generated based on the module, application and resource descriptors. As shown in the figure, application  $A_1$  requires sub-application  $A_2$  and module  $M_1$ . Application  $A_2$ , by itself, requires  $M_2$  and  $M_3$ . Each module  $M_i$  needs a set of required resources which are shown as single node  $R_i$ . Node  $C_i$  represents each machine of the cluster. There is an arc between each node  $R_i$  and  $C_j$  if and only if  $C_j$  fulfills all the required resources of  $R_i$ . For example,  $C_1$  and  $C_2$  meet requirements in  $R_2$ . Hence the relevant links are created among them.

Dependency graph  $\mathbb{G} = (N, A)$  is a directional graph where  $N$  is a finite set of nodes  $n_i$  and  $A$  is a finite set of arcs from  $n_i$  to  $n_j$ . Each node  $n_i \in N$  belongs to one of the subset  $N_c$ ,  $N_d$  or  $N_t$  where

- $N_c \subseteq N$  is a subset of conjunctive nodes which implies that each  $n_i \in N_c$  depends on all of its successor nodes.
- $N_d \subseteq N$  is a subset of disjunctive nodes which implies that each  $n_i \in N_d$  depends on one of its successor nodes.
- $N_t \subseteq N$  is a subset of terminal nodes or leaves of  $\mathbb{G}$ .

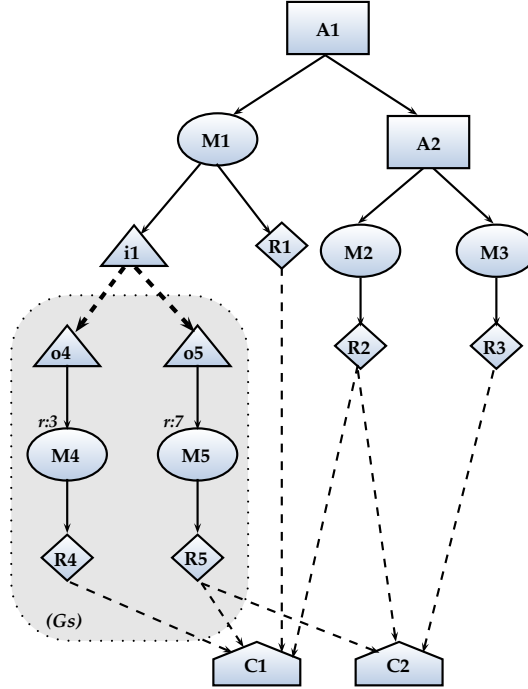


Figure 7.4: An example of graph used for dependency resolving.  $A_i$  indicates an application.  $M_i$  is a module,  $R_i$  and  $C_i$  are respectively required resource and computer,  $o_i$  and  $i_i$  are respectively output and input of a module.

Each node  $n_i$  is properly labeled as either *Application*, *Module*, *RequiredResource*, *Computer*, *Input* or *Output*.

In addition, system can automatically resolve dependencies across modules based on data they required (input) and produce (output). If *AutoDependency* property of  $M_i$  is enabled, a sub-graph  $\mathbb{G}_s \subseteq \mathbb{G}$  is generated for each required input  $i_i$  of  $M_i \in \mathbb{G}$ .  $\mathbb{G}_s$  includes all modules  $M_j$  which their output  $o_j$  has the same *DataType* as  $i_i$ . In Figure 7.4, module  $M_1$  require a specific data (e.g. 3D position data) from its input  $i_1$  which can be produced by either  $M_4$  or  $M_5$ . Hence, these modules are added to  $\mathbb{G}$  and the relevant links from  $i_1$  to  $o_4$  and  $o_5$  are created.

### 7.4.2 Dependency resolving and load balancing

Dependency resolving and load balancing involve finding required set of modules and their connections to run on proper set of clustered machines. Solution  $\mathbb{S}_{nodes}(n_i) \subseteq N$  is a minimum subset of nodes which satisfy dependencies of  $n_i$  and consecutively all its successors.  $n_i \in N_c$  is considered as *satisfied* if and only if all its immediate successors are included in the solution. Similarly,  $n_i \in N_d$  is considered as *satisfied* if at least one of its immediate successors is included in the solution. In Figure 7.4,  $N_d = \{i_1, R_{1..5}\}$ ,  $N_t = \{c_1, c_2\}$  and the rest belong to  $N_c$ .

There are some issues should be dealt with when searching for a solution. For example, while resolving dependencies of disjunctive node  $i_1$ , a proper decision should be taken to choose between  $o_4$  or  $o_5$ . Moreover, if  $o_4$  is chosen, the relevant connection from  $i_1 \rightarrow o_4$  should be added to the solution which later will be used by deployer to interconnect modules  $M_1$  and  $M_4$ . Therefore, we define solution  $\mathbb{S} = \langle \mathbb{S}_{nodes}, \mathbb{S}_{cons} \rangle$  where  $\mathbb{S}_{cons}$  is a list of required connection  $n_i \rightarrow n_j$  and  $n_i, n_j \in \mathbb{S}_{nodes}$ .

Formalizing the problem using dependency graph, breaks down the complexity of solution finding into graph exploration where different standard methods can be employed. Algorithm 1 demonstrates a recursive approach to find solution  $\mathbb{S}$  from the given node  $n_s$ . Line 1, initializes  $N_{suc}$  with a subset of immediate successors of  $n_s$  which are not *satisfied*. Lines 2 – 4 constraints that every leaf of graph must be of type *Computer*. Lines 5 – 20 propagates the algorithm in  $N_{suc}$ , taking proper decision while facing a conjunctive or disjunctive node. If  $n_s$  is a conjunction, all its successors are explored and added to the solution. On the other hand, if  $n_s$  is a disjunction,  $bestOf(N_{suc})$  chooses the proper successor node  $n_i$  to be explored. If  $n_s$  is of type *Input*, first call to  $bestOf(N_{suc})$  will return a successor which its corresponding *Module* has the highest rank. For example, in Figure 7.4,  $bestOf(\{o_4, o_5\})$  chooses  $o_4$ . Consecutive calls to  $bestOf$  will return the second-highest rank and etc. If  $n_s$  is of type *Computer*,  $bestOf(N_{suc})$  behaves as load balancer and selects the computer with less load average. Whenever required, line 19 add the necessary connection (i.e. from an input to output) to  $\mathbb{S}_{cons}$ . Line 22 adds the current node  $n_s$  to the solution to be considered as *satisfied*. Finally the algorithm return true value if all the required dependencies



**Algorithm 1** DEPRESOLVER( $n_s$ )**Require:**  $\mathbb{G}, n_s \in N$ .**Ensure:**  $\langle \mathbb{S}_{nodes}, \mathbb{S}_{cons} \rangle$ .

---

```

1:  $N_{suc} = Successors(n_s) \setminus \mathbb{S}_{nodes}$ 
2: if  $n_s \in N_t$  and  $n_s \notin N_{computers}$  then
3:   return false
4: end if
5: if  $n_s \in N_c$  then
6:   for  $n_i \in N_{suc}$  do
7:     if not DEPRESOLVER( $n_i$ ) then
8:       return false
9:     end if
10:  end for
11: else if  $n_s \in N_d$  then
12:  repeat
13:     $n_i \leftarrow bestOf(N_{suc})$ 
14:  until not DEPRESOLVER( $n_i$ )
15:  if  $n_i = \emptyset$  then
16:    return false
17:  end if
18:  if  $n_s \in N_{inputs}$  then
19:     $\mathbb{S}_{cons} \leftarrow \mathbb{S}_{cons} \cup \{ \langle n_s, n_i \rangle \}$ 
20:  end if
21: end if
22:  $\mathbb{S}_{nodes} \leftarrow \mathbb{S}_{nodes} \cup \{n_s\}$ 
23: return true

```

---

are satisfied\*.  $\mathbb{S}_{nodes}$  includes every type of nodes. Therefore, the final step is to select nodes labeled as *Module* and their corresponding machine which are labeled as *Computer* from  $\mathbb{S}_{nodes}$ .

---

\*Detail explanation of load balancing and proof of the algorithm is out of the scope of this work. However, readers are motivated to apply different decision making and load balancing mechanisms.

## 7.5 An example in YARP

To demonstrate how our proposed approach can be adopted by distributed robotic frameworks, we briefly describe the implementation of *Yarpmanager* over YARP. Through this, we explain how the abstract layers shown in Figure 7.3 and our generic conceptual model can greatly facilitate adaptation procedure to desired robotic frameworks.

YARP is a multi-platform distributed robotic middleware which consists of a set of libraries, communication protocols, and tools to keep modules and devices cleanly decoupled. Based on Observer pattern, communication in YARP allows the state of special port objects (which is named such as */port*) to be delivered to any number of observers, in any number of processes which are distributed over different machines. Modules written using YARP libraries, are compiled into runnable programs which can be launched on different machines using a special service (i.e. *yarprun*) and interconnected from an external program (e.g. another YARP program, the shell or shell script).

The implementation procedure of our proposed approach in YARP mostly involves two general phases: 1) Defining a standard to store and load the description of each entity; 2) Implementing a proper module launcher and interconnector. In our implementation, we use XML language to describe and store the information of each module, application and resource. Listing 7.1 exemplifies description of the resources used in Figure 7.4. It simply defines two computers  $C_1$  and  $C_2$  where  $C_1$  is equipped with a specific GPU.

```

1 <resources>
2   <computer>
3     <name>C1</name>
4     <gpu>
5       <name>Tesla C1060</name>
6       <capability>1.3</capability>
7     </gpu>
8   </computer>
9   <computer>
10    <name>C2</name>
11  </computer>
12 </resources>

```

Listing 7.1: Resource description in YARP.

Module  $M_1$  is also described in Listing 7.2. It shows that  $M_1$  requires an input data of type 3D-Position from its port  $i_1$ . Moreover, to operate properly, it needs a GPU which its computational capability is higher than 1.2.

```

1 <module>
2   <name>M1</name>
3   <input>
4     <type>3D-Position</type>
5     <required>yes</required>
6     <priority>no</priority>
7     <port>/i1</port>
8   </input>
9   <dependencies>
10    <computer>
11      <gpu>
12        <capability opt="higher">1.2</capability>
13      </gpu>
14    </computer>
15  </dependencies>
16 </module>

```

Listing 7.2: Module description in YARP.

The description of application  $A_1$  in Listing 7.3 states that  $A_1$  needs module  $M_1$  and another application  $A_2$ . Using `<node>`, it is possible to specifically indicate on which computer runs  $M_1$ . In this example, this field is left empty to let Algorithm 1 automatically selects the proper computer from those which are listed in Listing 7.1. It is

also possible to indicate which module launcher should be employed to execute the module. In the same way, the description of  $A_2$  can be provided in another XML file. Detailed explanation of XML template files and the standard can be found in [65].

```

1 <application>
2   <name>A1</name>
3   <module>
4     <name>M1</name>
5     <node></node>
6     <autodependency>yes</autodependency>
7     <deployer>yarprun</deployer>
8   </module>
9 </application>
10 <application>
11   <name>A2</name>
12 </application>

```

Listing 7.3: Application description in YARP.

The Deployment layer in YARP, is implemented using yarprun. It is a distributed client–server tool which receives the necessary information for running modules from remote clients and launch them on the server’s host. Deployment layer provides essential routines to start, interconnect and manage the modules by communicating with yarprun. Using these routines, Supervisor can robustly execute, interconnect and monitor modules in separate state–machines; this also allows it to take the proper decision in case of failures (module crash or termination). Moreover, using yarprun, the Deployment layer also offers to the Resource discoverer routines for collecting the status of the machines in the cluster (CPU load, memory usage, etc.).

## 7.6 Conclusions

This chapter has introduced a systematic approach to enhance scalability and management of software modules in distributed robotic architectures which results from our long–run experiences in developing humanoids robotic applications. We have discussed some practical difficulties which developers usually encounter for managing large number of modules distributed over different machines. Dependency resolving

of collaborative software components, heterogeneous modules deployment and portability of robotic applications are some issues which are mostly left in the background by robotic researchers. We have formalized the problem using our conceptual model which might not be totally general but still fairly comprehensive to be employed by many robotic frameworks. We have proposed a specific graph exploration algorithm to deal with the dependency resolving and load balancing. A example in YARP has been presented to demonstrate how our multilayer architectural design facilitates adaptation of the approach to any robotic framework.

# Chapter 8

## Conclusions

### 8.1 Contribution

This research study investigated the application development requirements in robotics and proposed some approaches to enhance reusability of the software components in robotic frameworks. The work has been divided in the three parts. The first part addressed the coordination problem of the modules in distributed architecture. The second part investigated how the extra requirement specific to an application can be added to an existing module as an extensible functionality. Finally, the last part dealt with the composition of modules in an application, their deployment and the implemented tools to support the application building and execution. The following presents the contribution of this research work in different parts.

#### 8.1.1 Coordination of components in distributed architectures

We have introduced a coordination mechanism for a network of behaviors based on port arbitration. We have shown that our approach allows to implement a non-trivial behavior that involves a sequence of actions. We have shown that the final behavior can be incrementally built as a composition of existing, simpler behaviors. Our approach is also fully distributed and minimizes the additional links required to perform arbitration.

We tested the behavior under different conditions and demonstrated that the resulting behavior is intrinsically robust and reactive to unexpected changes in the environment.

We have also illustrated a mechanism for modeling and coordinating behaviors based on our port arbitration approach. We have shown how robotic tasks can be represented using our behavioral description model and coordinated in a distributed component-based framework without any central coordinator. Remarkably, we demonstrated that in our framework, based on different behavioral descriptions, several robotic applications can be implemented using the same reusable software components.

### **8.1.2 Extensibility and reusability of components**

We have introduced the port plug-in system to extend modules functionalities by adding scripts to the ports that allow data monitoring, filtering, transforming and arbitration. Based on our approach we have developed specific plug-ins for port's data monitoring and arbitration to alleviate the problem of coordination and facilitate the development of reusable components.

The main advantage of our approach is that it allows to limit application specific functionalities to scripts that are external to the modules which can be added and executed at runtime. This maintains modules clean from unnecessary complexity and enhances their reusability. Finally, by using embedded scripts inside the ports, we can avoid introducing specific modules to achieve the required functionalities, thus, reducing communication and deployment overhead.

### **8.1.3 Composition and deployment of components**

We have illustrated a systematic approach to enhance scalability and management of software modules in distributed robotic architectures and discussed some practical difficulties which developers usually encounter when managing large number of modules distributed over different machines. Dependency resolving of collaborative software components, heterogeneous modules deployment and portability of robotic applications are some issues which are mostly left in the background by robotic researchers.

To address these issues, we have formalized the problem using our conceptual model which might not be totally generic but it is still fairly comprehensive to be employed by many robotic frameworks. We have proposed a specific graph exploration algorithm to deal with the dependency resolving and load balancing.

#### 8.1.4 Software and Tools

The coordination mechanism using port arbitration has been developed and completely integrated in the YARP framework as a new carrier recognized as *priority\_carrier*. The port plug-in system has also been implemented in form of another carrier (*portmonitor\_carrier*) which allows to access, modify and arbitrate data using LUA scripting language. The source code and the relevant examples demonstrating different potential usages of the port plug-in system can be found in the YARP repository.

Some essential tools for robotic application development and execution have been also developed in the YARP framework. These tools support the major component-based software development cycle: *i*) configuration, composition and coordination of the modules (the “gyarpbuidier”) and *ii*) deployment and monitoring (the “gyarpmanager”). The software and tools have been developed under GNU General Public License (GPL) and can be freely downloaded from the YARP repository along with the relevant documentation.

## 8.2 Discussion and Future works

There is still no emerging consensus over a generic robot programming paradigm. The decision to choose between explicit and implicit representation of the knowledge or between model-based and model-free highly depends on the research context and its objectives. Coordination of software components in a distributed architecture usually adds considerable overhead to the robotic application design process and often pulls the development of software components in specific direction. That can make significant impact on the reusability and viability of software components. Thus, solutions



offered by a paradigm might be still suboptimal in other contexts. However, the software architecture techniques and frameworks should limit the downside effect of each paradigm on the reusability of the software components.

We have introduced a coordination mechanism which makes use of existing data connections among the components and arbitrates them based on their activation status. The definite advantage of the proposed coordination mechanism is the simplicity and its generality over different robotic middlewares since the concept of the data-flow ports, has become a de facto standard in distributed software frameworks. Despite many robotic applications can be designed using port arbitration within the context of behavior-based system, the approach can suffer loss of information for coordination which is not made available in the arbitrator. A better solution to this problem is to use the port plug-in system which allows to define the coordination logic (arbitration rules) by also inspecting the data transmitted through the connections. The proposed coordination domain is not limited to the data-driven systems nor to data arbitration in the input ports. The same approach can be also used in control-driven architectures by arbitrating separate event messages which drive the coordination of the components.

Even though coordination based on port arbitration can cover a wide variety of robotic application developments, we have experienced certain limitations in the system. First, communication of software components with the external world is not limited via its streaming ports. Since arbitration is usually done on the data from unidirectional connection to an input port, it cannot be easily used in a service-oriented system where interactions between modules is bidirectional and done using blocking remote procedure calls. Moreover, a robotic task might require performing a sequence of actions in different steps where actions in each step should be synchronized with the internal states of the components which implement the corresponding actions. In other words, to perform an action, a series of commands should be given to a component in a specific order and its internal state should be monitored before sending a new command. This can be also made by port monitoring and arbitration, nevertheless, delegating this responsibility to a dedicated, external component can be preferable in favor of simplicity and performance.

The port arbitration approach can be used in the design of any robotic system where immediate reaction to changes in the system's state is required and these minor changes

are not necessarily needed to be reasoned about by a third-party component. For instance, minimizing delays in a distributed control application is functionally more important than making every change of the system's state explicitly visible to a separate coordinator via events. However, choosing between an explicit coordination mechanism and more reactive but less explicit way of orchestrating components, is a design choice and depends on functional requirements of robotics system. It is important to notice that the coordination approach using port arbitration does not constrain and limit the application designer to a specific mechanism for the orchestration of components. For example, the application designer can build part of the coordination using state machine and another part using port arbitration. The necessary functionalities to integrate these two parts can be provided using port plugin system.

The formalism for representation of the components and their configuration in applications, concentrates mostly on the necessary information for dependency resolving and deployment of the component in a distributed architecture. Most of the future work will be dedicated to extend the available formalism to support description of components at every level of composition. That is, software modules can be integrated and interconnected to build another component and new type of data can be also created by proper employment of port monitor plug-ins. In other words, a component designer can chose a mixture of off-the-shelf components and put them together. The port-arbitration mechanism (or other coordination) can be employed to coordinate these sub-modules. The designer can also implement some extra functionalities (e.g., data filtering and conversion) using port plug-in system and specify which interfaces should be exposed by the final component. All the process can be done using proper tool (i.e., by extending the "gyarpbuilder" to support the new formalism) and without any modification to the subsystems. Another important issue which will be investigate in the future work is the behavioral verification of the components and applications. This should be done by extending the current formalism and the corresponding tools (e.g., the "gyarpbuilder") to employ a suitable formal model in the literature for verification of behavior of the system.

# Appendix A

## Tools

This appendix shortly presents some tools, which have been developed with the goal of facilitating the application development in YARP framework. The tools support the major component-based software development cycle: *i)* the configuration, composition and coordination of the modules ( the “gyarpbuidier”) and *ii)* the deployment and monitoring (the “gyarpmanager”). They are developed under GNU General Public License (GPL) and can be freely downloaded from the YARP repository at <https://github.com/robotology/yarp.git>.

### A.1 The gyarpmanager

The command-line utility, the “yarpmanager”, and its graphical companion, the “gyarpmanager”, are a set of deployment tools for running and managing multiple programs on a set of machines. The implementation partially uses YARP framework and fully supports Windows, Linux and Mac Os.

The prominent features of the tool-set are as follows:

- Running, stopping, killing and monitoring multiple programs on the current or remote machines.

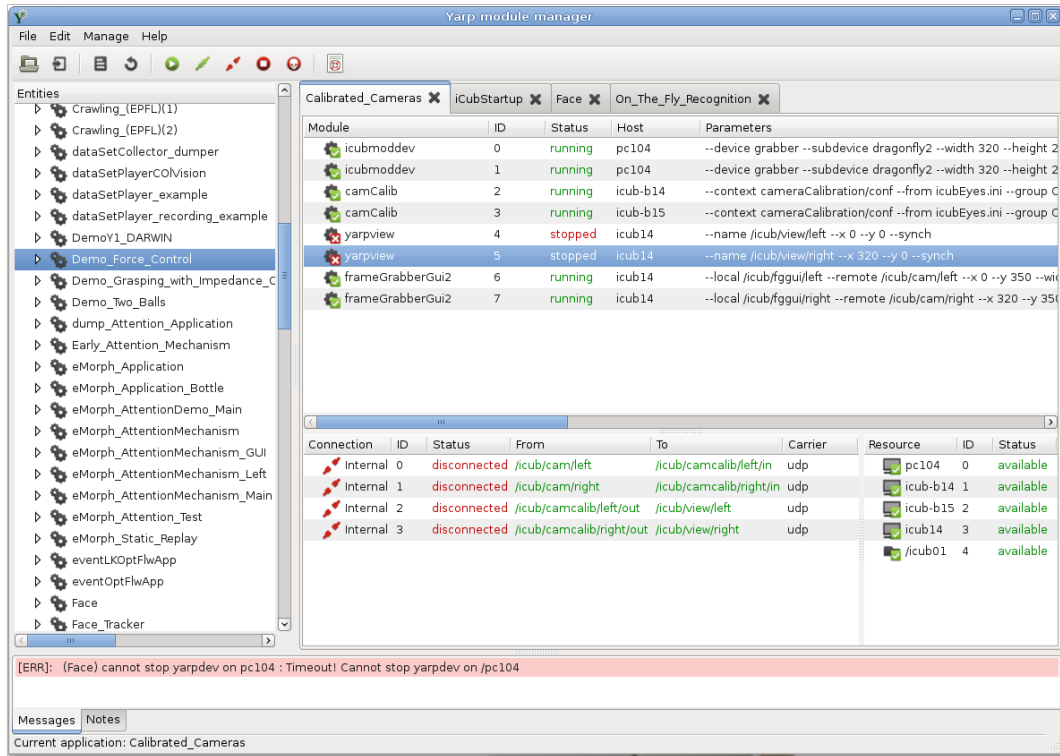


Figure A.1: A screenshot of gyarpmanager.

- Supporting different customizable deployer such as “yarprun”, “yarpdev”, “local-broker”, “script-broker”.
- Establishing ports connections manually and automatically.
- Managing multiple programs which are grouped as different applications.
- Running programs concerning their dependencies.
- Recovering programs from failure.
- Automatically assigning programs to proper machines using load balancing and smart resource discovery mechanism to improve their execution performance.
- Discovering information and status of machines in cluster of computers (e.g., hardware, platform, CPU load).

- Enabling users to specify the resource dependencies of the program using module description file.
- Enabling users to provide information about specific devices (e.g GPU) on a machine using resource description file.
- Enabling users to build an application which is composed of multiple sub-applications.
- Remapping ports name at run time using port prefix mechanism.
- Inspecting and monitoring data of the connections using YARP standard tools such as “yarp read”, “yarpview”, “yarphear” and “yarpscope”.

## **A.2 The gyarpbuilder**

The “gyarpbuilder” is a cross-platform (supports Windows, Linux and Mac Os) graphical tool for rapid application development in YARP framework. It enables the user to easily develop an application by configuring and interconnecting the available modules.

The tool makes use of the YARP module descriptions (as described in Chapter 7) and represents them in a graphical way. To build a new application, a developer can put the modules together, configure and interconnect them. The “gyarpbuilder” also performs some simple model checking and warns the user if some of the constraints such as required input connections or parameters for a module are not satisfied.

The port arbitration mechanism (see Chapter 3) can be also used for coordination of the corresponding components during the application development using the “gyarpbuilder”. An application developer can add an arbitrator entity to any input port and easily configure it with the necessary rules for the arbitration. Whenever a new rule is added to the arbitrator, a consistency check on the rules is performed to guarantee that only a single connection can deliver data at any given time.

Using the provided resource description of the available machines in a cluster (see Chapter 7), the deployment information can be manually set for the execution of the

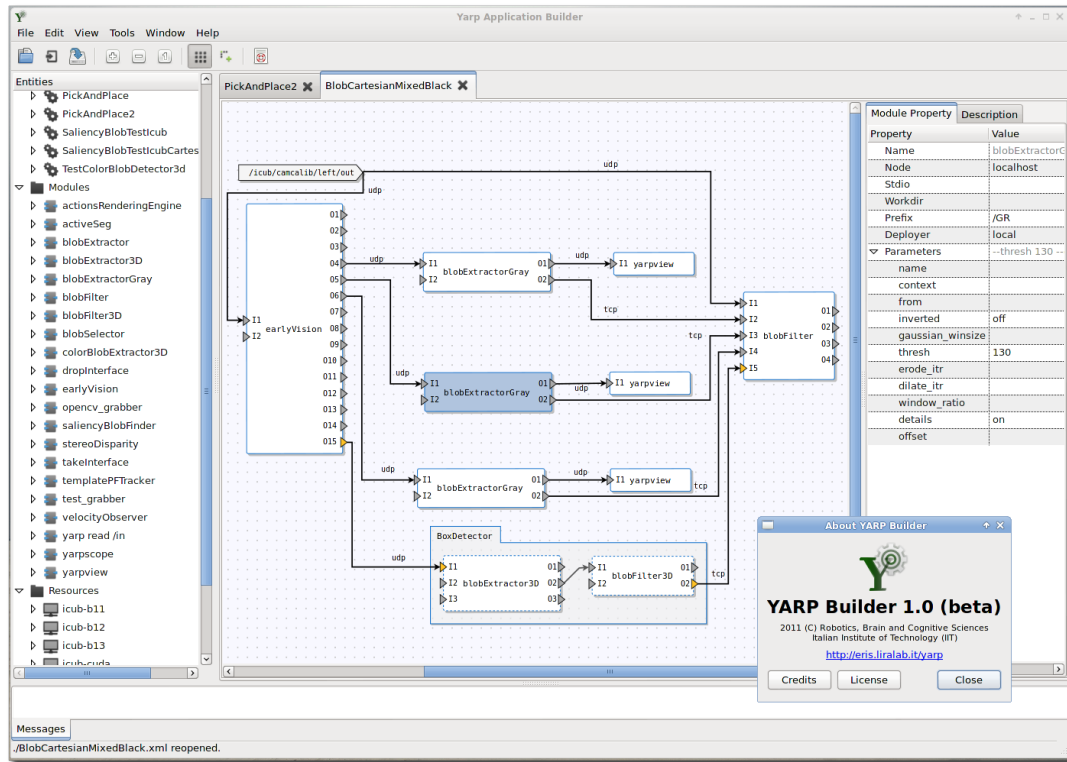


Figure A.2: A screenshot of gyarpbuilder

modules or they can be configured to be deployed using the automated load balancer. Eventually, the application can be loaded and launched using the “gyarpmanager”.

# Bibliography

- [1] FxEngine: A Framework for data flow processing and the design of dynamic systems using plugins. 66
- [2] Pluma: an open source C++ framework for plug-in management. 66
- [3] ZeroMQ: The intelligent transport layer. <http://www.zeromq.org>. 65
- [4] Hugo and others Klotzbuecher, Markus and Hochgeschwender, Nico and Gherardi, Luca and Bruyninckx, Herman and Kraetzschmar, Gerhard and Brugali, Davide and Shakhimardanov, Azamat and Paulus, Jan and Reckhaus, Michael and Garcia. The BRICS Component Model : A Model-Based Development Paradigm For Complex Robotics Software Systems Categories and Subject Descriptors. *28th ACM Symposium on Applied Computing (SAC), Coimbra, Portugal, 2013.* 2
- [5] N Ando, T Suehiro, and T Kotoku. A software platform for component based rt-system development: OpenRTM-Aist. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98, 2008. 65, 78, 80
- [6] Farhad Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 19, 1998. 12
- [7] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. 12

- [8] C Armbrust, M Proetzsch, B H Schäfer, and K Berns. A Behaviour-based Integration of Fully Autonomous, Semi-autonomous, and Tele-operated Control Modes for an Off-road Robot. In *Proceedings of the 2nd IFAC Symposium on Telematics Applications, Timisoara, Romania*, 2010. 14, 31
- [9] T Asfour, DN Ly, K Regenstein, and R Dillmann. Coordinated task execution for humanoid robots. *Experimental Robotics IX*, 2004(June):18–21, 2006. 8, 9
- [10] Howard Barringer, Michael Fisher, Dov Gabbay, Graham Gough, and Richard Owens. METATEM: A framework for programming in temporal logic. In *Step-wise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 94–129. Springer, 1990. 11
- [11] M Belguidoum and F Dagnat. Dependability in software component deployment. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX'07. 2nd International Conference on*, pages 223–230. IEEE, 2007. 82
- [12] J Bohren and S Cousins. The SMACH high-level executive. *Robotics & Automation Magazine, IEEE*, 2010. 7
- [13] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007. 11
- [14] RA Brooks. Intelligence without representation. *Artificial intelligence*, 47(1):139—159, 1991. 10, 14, 15
- [15] Davide Brugali and Patrizia Scandurra. Component-based Robotic Engineering Part I : Reusable building blocks. XX(4):1–12, 2009. 1, 13, 33, 63, 65
- [16] Davide Brugali and Azamat Shakhimardanov. Component-based Robotic Engineering Part II : Systems and Models. XX(1):1–12, 2010. 46, 65
- [17] 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. 1, 13, 65, 78



- [18] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100:677–691, 1986. 22
- [19] E Caron, P K Chouhan, H Dail, and Others. GoDIET: a deployment tool for distributed middleware on Grid’5000. 2006. 81, 82
- [20] T H J Collett, B A MacDonald, and B Gerkey. Player 2.0: Toward a Practical Robot Programming Framework. 2003. 78
- [21] Hugo Costelha. Robotic Tasks Modeling and Analysis Based on Petri Nets. *users.isr.ist.utl.pt*, 2003. 9
- [22] Hugo Costelha and Pedro Lima. Robot task plan representation by Petri nets: modelling, identification, analysis and execution. *Autonomous Robots*, 33(4):337–360, March 2012. 9
- [23] David Cross. *Data munging with Perl*. Manning Publications, 2001. 63
- [24] Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008. 11
- [25] Larry Dragich. The Anatomy of APM, 2008. 63
- [26] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003. 79
- [27] A Flissi, J Dubus, N Dolet, and P Merle. Deploying on the Grid with DeployWare. In *Cluster Computing and the Grid, 2008. CCGRID’08. 8th IEEE International Symposium on*, pages 177–184. IEEE, 2008. 81, 82
- [28] A Flissi and P Merle. A generic deployment framework for grid computing and distributed applications. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 1402–1411, 2006. 81
- [29] G Frey and M Minas. Editing, visualizing, and implementing signal interpreted petri nets. *Proceedings of the AWPN 2000*, 2000. 9

- [30] Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004. 66
- [31] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985. 12
- [32] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992. 12
- [33] Michael Georgeff, Barney Pell, and Martha Pollack. The belief-desire-intention model of agency. *Intelligent Agents V*, 1999. 11
- [34] C D Gill and W D Smart. Middleware for robots. In *Intelligent Distributed and Embedded Systems, Papers from the 2002 AAAI Spring Symposium*, Gaurav S. Sukhatme and Tucker Balch (Ed.), pages 1–5, 2002. 78
- [35] M Giorgio, F Paul, and N Lorenzo. Towards Long-Lived Robot Genes. *Elsevier*, 2007. 65
- [36] D Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987. 6
- [37] Eiji Hayashi, Kei Ueyama, and Motoki Yoshida. Autonomous action selection with motivation-based consciousness and behavior architecture of animal. *The 5th International Conference on Automation, Robotics and Applications*, pages 294–299, December 2011. 15
- [38] George T Heineman and William T Council. Component-based software engineering: putting the pieces together. 2001. 1
- [39] Roberto Ierusalimsky, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996. 52
- [40] A Ikezoe, H Nakamoto, and M Nagase. Development of RT-middleware for image recognition module. In *SICE-ICASE, 2006. International Joint Conference*, pages 2036–2041. IEEE, 2006. 78

- [41] ZeroC Inc. Internet Communications Engine. <http://zeroc.com/ice.html>. 65
- [42] C Jang, S I Lee, S W Jung, B Song, R Kim, S Kim, and C H Lee. Opros: A new component-based robot software platform. *ETRI journal*, 32(5):646–656, 2010. 65
- [43] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003. 8
- [44] C Kertész. Dynamic behavior network. *Applied Machine Intelligence and Informatics*, pages 207–212, 2012. 10, 17, 33
- [45] M Klotzbücher and Herman Bruyninckx. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *Software Engineering for Robotics*, 1(January):28–56, 2012. 7, 47
- [46] S Lacour, C Pérez, and T Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 284–287. IEEE Computer Society, 2005. 81
- [47] Arne Lehmann, Ralf Mikut, and Tamim Asfour. Petri nets for task supervision in humanoid Robots. *VDI BERICHTE*, 2006. 9
- [48] Joaquin López, Diego Pérez, and Eduardo Zalama. A framework for building mobile single and multi-robot applications. *Robotics and Autonomous Systems*, 59(3-4):151–162, March 2011. 9
- [49] Jonathan M MacKenzie, Douglas C and Arkin, Ronald C and Cameron. Multi-agent mission specification and execution. *Robot colonies*, pages 29–52, 1997. 16
- [50] P Maes. How to do the right thing. *Connection Science*, 1989. 15
- [51] Fulvio Mastrogiovanni, Ali Paikan, and Antonio Sgorbissa. Semantic-Aware Real-Time Scheduling in Robotics. *IEEE Transactions on Robotics*, 29(1):118–135, February 2013. 54

- [52] Maja J Mataric. Behavior-based robotics. Technical report. 36
- [53] Giorgio Metta, G Sandini, and David Vernon. The iCub humanoid robot: an open platform for research in embodied cognition. *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pages 50—56, 2008. 15, 32, 67
- [54] L. Metta, G. Fitzpatrick, P. and Natale. YARP: Yet Another Robot Platform. *International Journal on Advanced Robotics Systems*, 3(1):43–48, 2006. 1, 13, 15, 28, 32, 51, 78, 79
- [55] Giulio Milighetti. Dynamic decision making for humanoid robots based on a modular task structure. 2009. 9
- [56] P Mohammadpour, M Sharifi, and A Paikan. A self-training algorithm for load balancing in cluster computing. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 1, pages 104–109. IEEE, 2008. 82
- [57] M Montemerlo, N Roy, and S Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2436–2441. IEEE, 2003. 9
- [58] T Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 1989. 8
- [59] Mario Murata, Kenichi and Horspool, R Nigel and Manning, Eric G and Yokote, Yasuhiko and Tokoro. Unification of active and passive objects in an object-oriented operating system. *Object-Orientation in Operating Systems, 1995., Fourth International Workshop on*, pages 68–71, 1995. 53
- [60] D Nakhaeinia, S H Tang, S B Mohd Noor, and O Motlagh. A review of control architectures for autonomous navigation of mobile robots. *Int. J. Phys. Sci*, 6(2):169–174, 2011. 14, 15
- [61] MN Nicolescu and MJ Mataric. Extending behavior-based systems capabilities using an abstract behavior representation. 2000. 33


- 
- [62] Torre Norte. Petri Net Models of a Robotic Task. (May), 2002. 8
- [63] OMG. Common Object Request Broker Architecture (CORBA/IIOP).v3.1. Technical report, OMG, January 2008. 65
- [64] R.T. Pack. *IMA: The intelligent machine architecture*. PhD thesis, 1998. 11
- [65] Ali Paikan. *(g)yarpmanager: a way of running and managing multiple programs on a set of machines*. 93
- [66] Ali Paikan, Giorgio Metta, and Lorenzo Natale. A port-arbitrated mechanism for behavior selection in humanoid robotics. *The 16th International Conference on Advanced Robotics*. 60
- [67] J D Pedersen. *Robust communications for high bandwidth real-time systems*. PhD thesis, Citeseer, 1998. 78
- [68] P Pirjanian. Behavior coordination mechanisms-state-of-the-art. *Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep. IRIS-99-375*, 1999. 10, 16
- [69] T J Prescott. Action selection. *Scholarpedia*, 3(1):2705, 2008. 16
- [70] M Proetzsch, T Luksch, and K Berns. Development of complex robotic systems using the behavior-based control architecture iB2C. *Robotics and Autonomous Systems*, 58(1):46–67, 2010. 10, 16, 17, 33, 78
- [71] 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. *Science*, 2009. 7, 65, 78
- [72] Anand S Rao. AgentSpeak (L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996. 11
- [73] Julio K. Rosenblatt. DAMN: a distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):339–360, April 1997. 16, 33

- 
- [74] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 2, 65, 66
- [75] M Scheutz and V Andronache. Architectural mechanisms for dynamic changes of behavior selection strategies in behavior-based systems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2377–2395, 2004. 10, 16
- [76] Azamat Shakhimardanov, Nico Hochgeschwender, and Gerhard K. Kraetzschmar. Component models in robotics software. *Proceedings of the 10th Performance Metrics for Intelligent Systems Workshop on - PerMIS '10*, page 82, 2010. 48
- [77] W D Smart. Is a Common Middleware for Robotics Possible? *Science*. 78, 82
- [78] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002. 1
- [79] V Tikhonoff, U Pattacini, L Natale, and G Metta. Exploring affordances and tool use on the iCub. *IEEE-RAS International Conference on Humanoid Robots*, 2013. 67, 69
- [80] Steve Tousignant, E Van Wyk, and Maria Gini. XRobots: A flexible language for programming mobile robots based on hierarchical state machines. *Robotics and Automation ( ... , pages 1773–1778, May 2012*. 7
- [81] T. Tyrrell. An Evaluation of Maes’s Bottom-Up Mechanism for Behavior Selection. *Adaptive Behavior*, 2(4):307–348, March 1994. 15
- [82] K Uhl and M Ziegenmeyer. MCA2-an extensible modular framework for robot control applications. In *Proceedings of CLAWAR, 10th International Conference on Climbing and Walking Robots*, 2007. 81
- [83] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995. 62

- 
- [84] B B Werger. Ayllu: Distributed port-arbitrated behavior-based control. In *In Proc., The 5th Intl. Symp. on Distributed Autonomous Robotic Systems*. Citeseer, 2000. 16
- [85] Reinhard Wolfinger and Deepak Dhungana. A component plug-in architecture for the. NET platform. *Modular Programming . . .*, pages 1–20, 2006. 66
- [86] BG Woolley and GL Peterson. Real-time behavior-based robot control. *Autonomous Robots*, (January):233–242, 2011. 14, 31
- [87] Matthias Zenger. *Programming language abstractions for extensible software components*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2004. 66
- [88] Jiaxiang Zhang. Selection and inhibition mechanisms for human voluntary action decisions. *NeuroImage*, July 2012. 16
- [89] Pouyan Ziafati, Mehdi Dastani, John-Jules Meyer, and Leendert van der Torre. Event-processing in autonomous robot programming. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems, AAMAS '13*, pages 95–102, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems. 48
- [90] VA Ziparo, L Iocchi, and D Nardi. Petri net plans: a formal model for representation and execution of multi-robot plans. *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 79—86, 2008. 9







ISTITUTO ITALIANO DI TECNOLOGIA  
DEPARTMENT OF ICUB FACILITY  
Via Morego, 30  
16163 Genova  
[www.iit.it](http://www.iit.it)

