UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences Appliquées
CODE - Computers and Decision Engineering
IRIDIA - Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

A CONTROL ARCHITECTURE FOR A HETEROGENEOUS SWARM OF ROBOTS

The Design of a Modular Behavior-based Architecture

Eliseo FERRANTE

Supervisor:
Prof. Marco DORIGO

Co-Supervisor:
Dr. Mauro BIRATTARI

Rapport d'avancement de recherche
Academic year: 2008/2009

Abstract

We propose a software architecture that can ease and speed up the development process of controllers for heterogeneous swarm systems. It is inherently modular and allows behaviors to be combined in layers and reused in multiple controllers. This can potentially speed-up the development process of controllers since they become much more readable and well structured. We validated the architecture through an experiment of collective navigation and obstacle avoidance using two different types of robots. The experiments show that, using the proposed architecture, complex behaviors can indeed be built through the combination of very simple behaviors.

Acknowledgments

First of all, I would like to thank my supervisor, prof. Marco Dorigo for giving me the possibility to work here at IRIDIA and to do research in Swarm Robotics. I would like to thank also dr. Mauro Birattari, for the priceless and countless insights he provided me so far. I'm indebted with prof. Andrea Bonarini, which referred me to Marco Dorigo and his wonderful lab. Furthermore, I acknowledge the financial support provided by the Swarmanoid project.

Thank you also to all my friends and colleagues here at IRIDIA, which have also provided an extremely valuable support into my professional and personal life. Amongst them, I would like to thank Nithigno and Carlotto, which have worked with me closely and without which this work woudn't have been accomplished. Thanks also to Marquito, for the unending support and feedback and for always remembering me which are the priorities during a PhD. Thanks also to Arnucci, Rehan, Ali, Antalo, Mattéo the Old and Mattèo the Bald, Manuilio, Franciesci (and his wonderful recipes!), Jeremie, Coligno, Giacomotto (and all the former visiting scientists that have passed through IRIDIA), Prasannotto, Ericco, Saifullah, Sabrigna, Gionni, Alex, Francisco, Thomas, Renaud, Christos (and all the others that have already the privilege to be callable *Doctor*), etc...etc...Also, I would like to thank all the colleaugues in the other labs involved in the Swarmanoid Project.

I would like to thank all my non-IRIDIAn friends that I've met here in Brussels, all my dear friends in Milano (hoping they've not forgotten me) and that moved away from it and all the others in Santeramo. To all those, I apologize for not including all their names here as they deserve, leaving with the promise they will be present in the PhD thesis, which is a much longer and hence suitable document :-).

Last but not least, I would like to thank my parents for supporting me during my studies that allowed me to be here, for the continuous support in my decisions, and for always a little less cloudy place I can always return to.

To all these and to all the others that I unfortunately left out due to the lack of space, THANK YOU!

Contents

List of Figures

Chapter 1

Introduction

This work deals with the design and the implementation of a software architecture that can be employed for the development of controllers for heterogeneous group of robots. With the proposed architecture, modularity and code reuse is made possible, yielding to a significant speed-up in the development of controllers.

The aim of this first chapter is to introduce the main scientific domains where this work should be placed, i.e. swarm intelligence and robotics. We then outline the goal of this document together with the structure of the entire document in the last section.

1.1 Swarm Intelligence and Swarm Robotics

Swarm intelligence (commonly abbreviated with SI) is a sub-field of artificial intelligence. It studies principles such as self-organization, decentralization and emergence of collective behaviors. The term was first introduced by Beni and Wang (1989) in the context of cellular robotic systems.

Swarm intelligence systems consist typically of a population of relatively simple agents which interact only locally with each other and with their environment, without global knowledge about their own state and of the state of the world. Furthermore, the agents follow often very simple rules and exhibit, to a certain degree, random interactions between each other. A recurrent property of these systems is the emergence of "intelligent" global behaviors, without centralized control/

Swarm intelligence is often inspired from the behavior of social insects and from other groups of animals. Examples include ant colonies (Detrain and Deneubourg, 2006), bird flocking (Reynolds, 1987), animal herding (Gautrais et al., 2007), colony of bacteria (Ben-Jacob et al., 2000), and fish schooling (Grünbaum et al., 2004).

The following are desirable properties of swarm intelligence systems:

- the system is composed of many, relatively homogeneous agents;

- the agents are relatively simple both in structure and in their behavior;
- interactions amongts agents are only local;
- control is fully distributed;
- the global behavior is not encoded in the system but emerges from the interaction amongst the agent and is more rich with respect to the behaviors of individual agents;
- the system is robust and adaptive with respect to unexpected event in the environment or in the system itself.

Swarm intelligence is composed of several sub-fields. Amongst them, we can find: swarm based optimization techniques like ant colony optimization (Dorigo and Stützle, 2004) and particle swarm optimization (Kennedy and Eberhart., 2001); data mining (Abraham et al., 2006); network routing (Caro and Dorigo, 1998); and swarm robotics (Dorigo and Şahin, 2004). Amongst all the sub-fields of swarm intelligence, the present work can be placed in swarm robotics.

Swarm robotics is a alternative approach to robotics which tackles some issues present in classical robotics. Such issues include the increase in complexity of the system as the task becomes more complex or the high sensitivity to faults. Swarm robotics consists in the application of principles of swarm intelligence to the field of robotics (Dorigo and Şahin, 2004).

Swarm robotic systems allow the development of cheaper, less complex robots, which are more robust. Other properties of the swarm robotic approach are flexibility, adaptability, redundancy and fault tolerance. For a review on swarm robotics, see Bonabeau et al. (1999) or Beni (2004). Examples of success applications of swarm robotics include flocking (Turgut et al., 2008), morphogenesis and self-assembly (Christensen et al., 2007; Groß and Dorigo, 2007), fault detection (Christensen et al., 2008), path formation and prey retrieval (Nouyan et al., 2008), coordinated behaviors (Sperati et al., 2008) and collective transport (Groß and Dorigo, 2008).

1.2 Robot Software Architectures

This work can also be placed in the domain of robot architectures. A *robot architecture* primarily refers to the software and hardware framework for controlling the robot. In particular, it concerns the development of software modules and the communication between them.

Robotic systems are complex and tend to be difficult to develop. They integrate multiple sensors and actuators, they might have many degrees of freedom and must reconcile hard real-time systems with systems which cannot meet real-time requirements. These architec-

tures, however, have so far been task and domain specific and have lacked suitability to a broad range of applications.

The first trend, during the 50s, was to produce architectures able to integrate reasoning and planning systems common to the mainstream AI systems of those times. A subsequent trend has focused on behavior-based or reactive systems. These systems are characterized by tight coupling between sensors and actuators, minimal computation, and decomposition of the problem which is based on behaviors themselves. When also behavior-based architectures reached their limits, the following step was to produce architectures that integrated planning with behavior-based control: the so-called hybrid architectures. However, this historical evolution (that will be described in more details in Section 3.1) has mainly concerned single robot systems.

1.3 Overall Goals

Although architectures for multi-robot systems have been developed (see Section 3.2), they do not immediately scale to swarm robotics systems. Hence, an interesting topic of research, we believe, is to investigate whether it is possible to develop an architecture where several individual-level behaviors can be orchestrated in order to guarantee the emergence of a global behavior capable of solving the task at hand. To achieve this, one can use the works described in Section 3.2.3 as an inspiration.

Furthermore, we believe that behavior-based architectures are a central cornerstone, i.e. they represent the essential lower layer for the development of individual level behavior that could then be integrated by the introduction of an extended architecture for swarm robotics. Hence, in Chapter 4, we introduce our work: the *behavioral toolkit*. The behavioral toolkit is a modular, behavior-based architecture developed for the Swarmanoid project (described in Chapter 2) in the context of swarm robotics. In this work, we will focus on the software aspect: we want our architecture to facilitate writing modular, effective and most importantly highly readable object oriented code describing behaviors, in order to speed up the efficiency of the development process. In Chapter 5 we test the proposed architecture on a heterogeneous swarm robotics task which involves the assembly of a robotic entity composed of two types of robots and its navigation in a complex environment with obstacles. We conclude the present document in Chapter 6 with final remarks and proposals of extensions to the proposed architecture.

Chapter 2

The Swarmanoid Project

This chapter is devoted to the introduction of the Swarmanoid project. It consists of a swarm robotics project that puts particular emphasis on the heterogeneity aspect. In this chapter, we first provide an introduction to the project as a whole. Subsequently, we briefly introduce the key components of the project, whose development is, at the time of writing, still on-going, i.e. the hardware and the simulation software.

2.1 Overview of the Swarmanoid Project

The creation of a flexible and autonomous robotics platform that is able to support humans in many of their activities has long been under research. Research in robotics has been progressing in different directions, amongst which we can find mobile robotics and humanoid robotics.

Mobile robotics has, as a key driving force in its development, the reduction of human intervention in potentially dangerous or unaccessible applications. Examples of such applications are cleanup of toxic waste, nuclear power plant decommissioning, planetary exploration, security, surveillance, etc...On the other hand, research in humanoid robotics has mainly been motivated by the need of supporting ordinary human daily activities. Assistance of elderly, sick or disabled people are key examples, as well as automation of ordinary but repetitive activities (mainly held indoor) such as receptionists and other tasks that require interaction with people.

Autonomous robotics can be also classified from another, orthogonal perspective. On one hand, a first approach is clearly to design and build a single robot, able to address a particular application. Such a robot would need to have all the capabilities in terms of sensors, actuators and control able to complete its task. Such an approach can be feasible for smaller-scale applications. However, as the complexity of the tasks increase, single robot approach is destined to not scale well enough due to the required complexity and cost.

An alternative approach is to design and build teams composed of multiple robots that

can autonomously and collectively solve the task. Potential advantages of using a distributed mobile robot systems include the reduction of the total cost, since each individual robot does not need to be over complex as in the single robot case. Also, designing a solution can be easier than in the case mentioned above. Other advantages include robustness, fault tolerance and parallelism. This means that, in a team of robots, a subset of them can take over a task that was not completed by other robots due to failure or inefficiency. Furthermore, multiple sub-tasks can be executed at the same time, thus increasing the whole system performance.

Swarm robotics is a very particular and peculiar sub-area of collective robotics. As already mentioned in Section 1.1, particularly important in swarm robotics are aspects such as self-organisation, decentralisation, local perception and communication.

Within the framework of swarm robotics, the *Swarmanoid Project* is a futuristic project that temporally and logically follows the *Swarmbots Project* (Dorigo et al., 2004; Mondada et al., 2004). As its predecessor, it is funded by the European Commission. Five research laboratories scattered through Europe are part of it: IRIDIA at Université Libre de Bruxelles in Belgium, Istituto di Scienze e Tecnologie della Cognizione at Consiglio Nazionale delle Ricerche in Italy, Laboratoire de Systèmes Robotiques and Laboratory of Intelligent Systems at Ecole Polytechnique Federale de Lausanne in Switzerland and Istituto Dalle Molle di Studi sull'Intelligenza Artificiale at Università della Svizzera italiana in Switzerland.

The Swarmanoid project is different with respect to most current studies in swarm robotic systems, which have focused on robotic swarms in which components are physically and behaviourally undifferentiated. In the Swarmanoid project, in fact, the focus is on how to design, realise and control a *heterogeneous* swarm robotic system capable of operating in a fully 3-dimensional man made, indoor environment. The swarm is composed of heterogeneous, dynamically connected, small autonomous robots of three types: eye-bots, hand-bots, and foot-bots.

The Swarmanoid project can be seen as an attempt to unify mobile swarm robotics with humanoid robotics. The vision is in fact the creation of some sort of *humanoid swarm*. The direction that is used to go towards humanoid swarming is specialization. The three robot types are introduced are each specialized on particular capabilities: eye-bots are specialized in perception and supervision, whereas hand-bots focus on manipulation and foot-bots on navigation and transport. More in particular:

Eye-bots fly or attach to the ceiling. Thanks to their positioning capabilities and to their camera, they are able to quickly explore the environment and locate targets, interesting objects or areas.

Hand-bots are intended to retrieve and manipulate objects located on walls, shelves, or tables. They can climb walls and obstacles by means of a rope they can launch in order to attach to the ceiling.

Foot-bots are wheeled robots equipped with a rigid gripper used to assemble with other foot-bots, transport hand-bots or target objects.

In the following sections we present the hardware capabilities of the robots. Finally, a simulation environment is also introduced. It has been developed to ensure rapid prototyping and testing of robot behaviors.

2.2 The Swarmanoid Hardware

In this section we briefly present the hardware of the robots of the Swarmanoid project. We first present the hardware capabilities that are shared between the robots, followed by the characterization each robot individually.

2.2.1 Common Devices

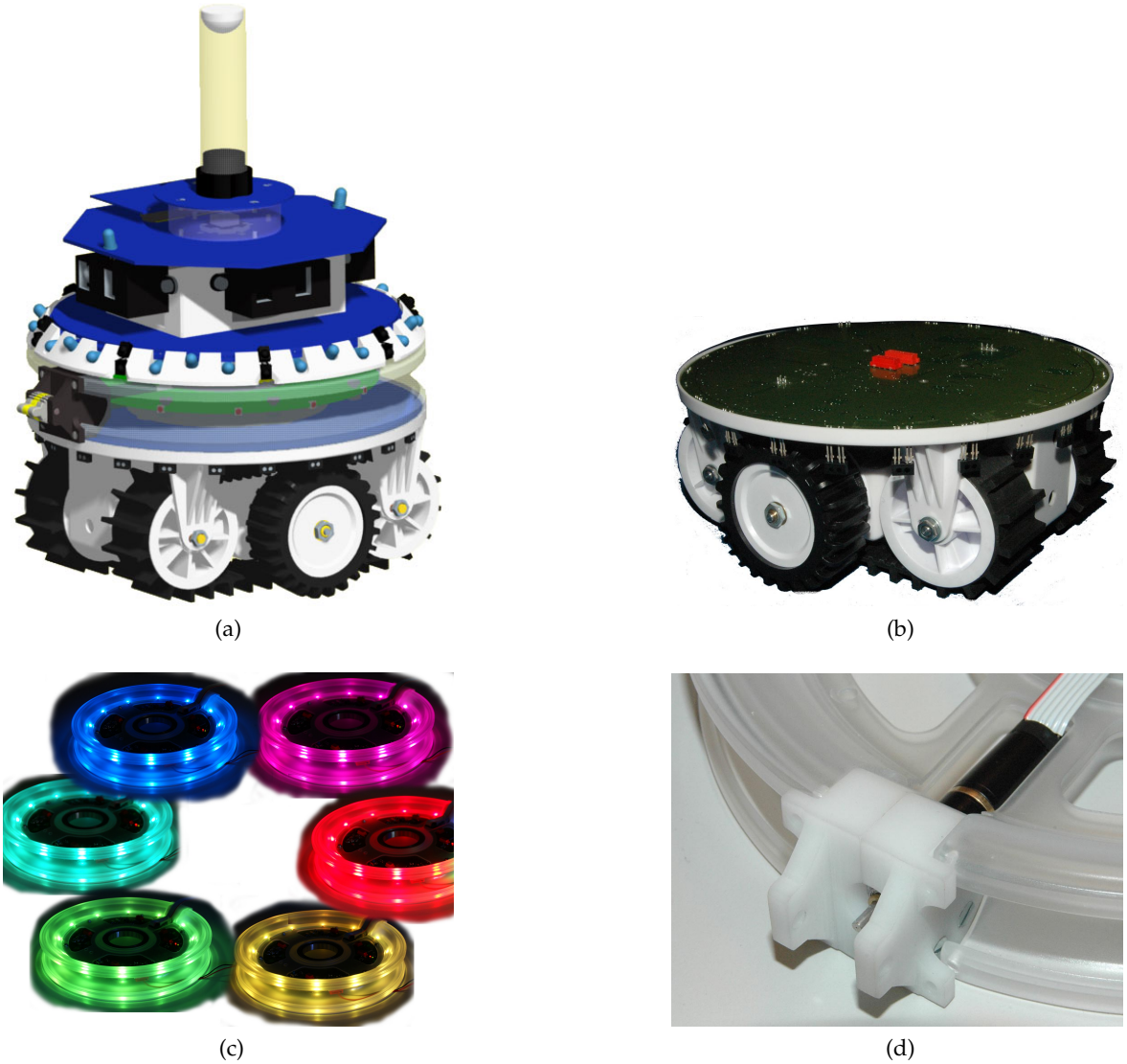
Some hardware capabilities are common to all robots. They consists ofn:

- The main processor board: the Freescale i.MX31 ARM 11 processor, a low-energy 533 MHz processor with a complete collection of subsystems such as USB host controller, integrated camera interface and SD card. The main board features 128 MB of DDR RAM and 64 MB of flash.
- The range and bearing subsystem, which is a module that allow local and situated communication in between the robots, achieved through the integration of radio and infrared technologies.
- The CAN bus, used to interconnect the different micro-controllers arbitrating each individual sensor/actuator module, and the main board.
- The Pixelplus 2.0 MegaPixels CMOS camera.

2.2.2 The Foot-bot

In the Swarm-bots Project, the *s-bot* provides the basic robotic platform. It consists in a small scale robot with many sensors and actuators. S-bots are able to connect to each other with a rigid gripper in order to form a *Swarm-bot*. The success of the Swarm-bots Project and especially of its s-bot suggested that the foot-bot design should be based on it. A more advanced prototype of the s-bot is the *marXBot*, a modular robot with higher computational power and more sensors with improved accuracy.

The foot-bot has been therefore based on the *marXBot* and Figure 2.1 details the some of its components: the overall design; the base of the robot that includes the wheels actuated by 2W motors; the LED ring; the gripper. Other key hardware components include: a set



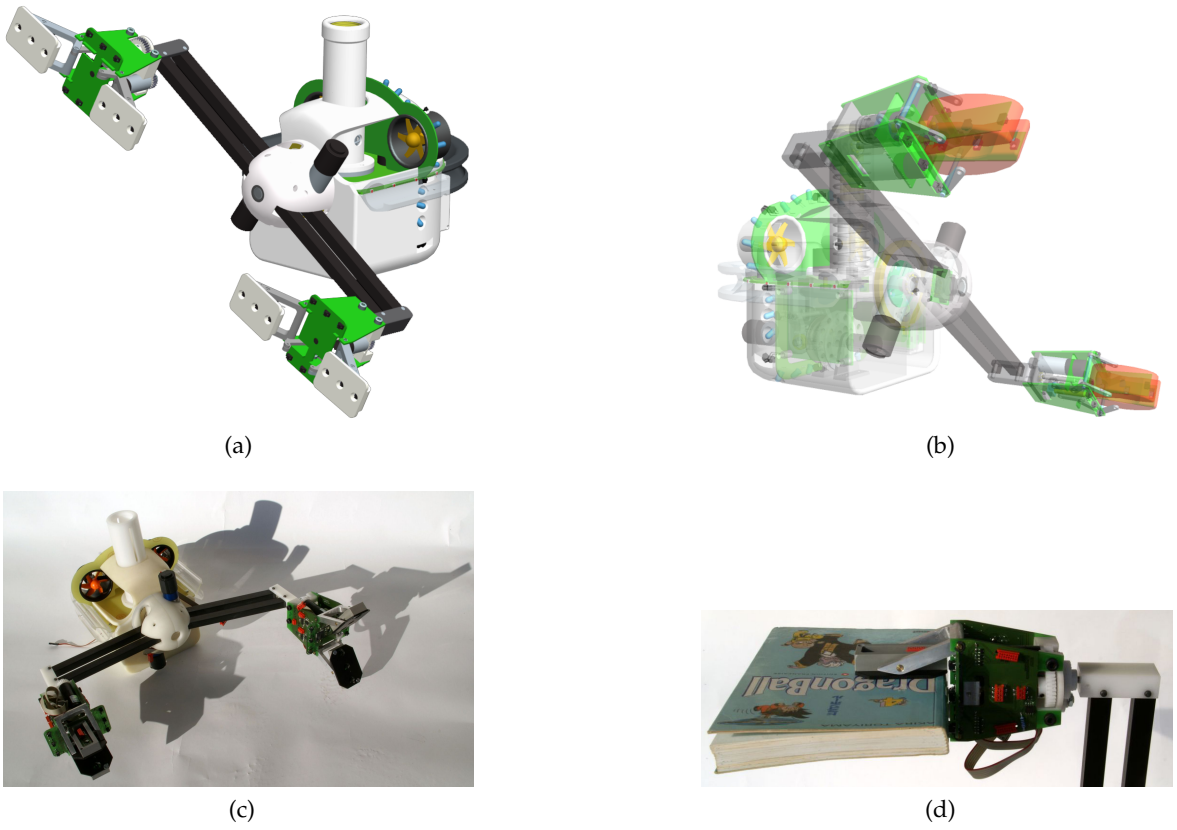
(a) (b)

(c) (d)

Figure 2.1: Foot-bot design: (a) the foot-bot CAD model, (b) the base of the foot-bot that is prototyped at the time of writing, (c) the foot-bot LED ring and (d) the foot-bot gripper

of 24 proximity sensors placed in a ring facing outside; 8 proximity sensors facing down; 4 contact ground sensors; a rotating long-range[1] infrared distance sensor; an omnidirectional camera and a camera looking up (whose common technology has already been described in 2.2.1). Additionally, the foot-bot has a hot-swappable battery, and a super-capacitor keeps the robot alive while the battery is being exchanged.

[1]Up to 1.5 m.



(a) (b)

(c) (d)

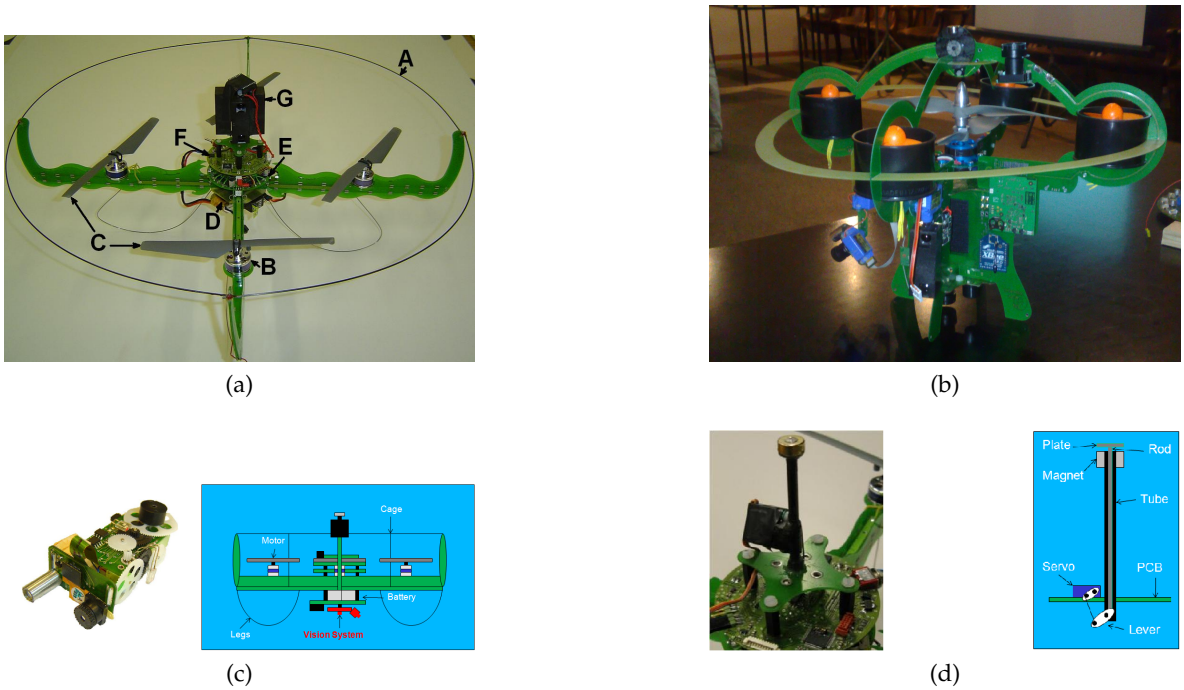
Figure 2.2: Hand-bot design: (a) the hand-bot CAD model, (b) a more detailed hand-bot CAD model, (c) the hand-bot prototype developed at the time of writing and (d) the hand-bot gripper grasping a book

2.2.3 The Hand-bot

The hand-bot represents a still unexplored robotic platform. It is designed to be a relatively low-cost, energy saving, light weight robot able to launch a rope, climb on vertical supports and manipulate objects located on the vertical plane (for example on shelves).

Figure 2.2 details some of the hand-bot's components. Other key components are: the rope launcher, which is made of 4 motors (one for recharging the spring and unlocking the launcher, one for the fast rewind of the cable when the magnet is detached, one for moving the hand-bot up and down the rope, and one servo-motor for connecting the previous motor to the rope driving system); the head, which is the attach and rotation point for the arms; a camera with a fish-eye lens placed in the head; two powerful motors to drive each arm independently and a third motor responsible for rotating the arms with respect to the body; the grippers, which are placed at the end of the arm and consist of two motors: one for rotation and one for grasping.

The hand-bot can also move on the ground thanks to foot-bots which can grip it. Climb-



(a) (b) (c) (d)

Figure 2.3: Eye-bot design: (a) one of the first prototypes, (b) one of the last prototype at the time of writing, (c) the eye-bot pan and tilt camera and (d) ceiling attachment mechanism

ing to reach objects located on shelves is achieved through a rope with magnetic attachment device. The rope is launched with a mechanical device allowing also the rewind of the cable. This requires the ceiling to be ferro-magnetic.

Manipulation of objects requires a more evolved gripper than the one mounted on the foot-bot. The current prototype is depicted in Figure 2.1d.

2.2.4 The Eye-bot

Similarly to the hand-bot, the design of a flying robot such as the eye-bot is very difficult. Its design and prototyping has been through a lot of phases (Figure 2.3a and 2.3b shows two of the prototypes of the robot). The eye-bot has the following capabilities: it has fully autonomous stability control including pitch, roll, yaw and altitude; the ability to hover and manoeuvre in an indoor environment; it can attach to the ceiling to save energy and to allow for a stable visual field of view; it has an omni-directional vision systemfor visual object detection and a visual colour signaling system for low-level communication between robots.

The robot is based on a quadrotor design. It will be equipped with an optic-flow sensor, which will be used to detect and avoid drifts in random directions.

The ceiling attachment device (Figure 2.3d) is located centrally at the top of the robot structure. Attachment to a ferro-magnetic ceiling is achieved by using a toroidal (passive)

magnet with an attractive force of 2.5kg. The detachment mechanism consists in a mechanical lever that pushes a carbon fiber rod through the center of the toroid.

The flight computer is located centrally at the top of the eyebot structure on the second level of the avionics board stack. It is in charge of all the low-level stability control and houses the critical stability sensors: gyros, accelerometers, pressure altitude, ultrasonic altitude and magnetometer.

The vision system 2.3c system is capable of panning 360 degrees in the horizontal plane and tilting 90 degrees from vertical to horizontal. It uses the same camera described in 2.2.1. A Red-Green-Blue (RGB) colour signaling light will be placed directly on the bottom of the eye-bot which can be used as a visual communication device between robots.

Finally, a omni-directional distance scanner is also available, which consists of two infrared triangulation sensors facing opposite directions.

2.3 The Swarmanoid Simulation Framework

In the context of the Swarmanoid project, a software simulation framework was also developed under the name of Autonomous Robots Go Swarming (ARGoS). By the term *simulation framework* we denote a set of tools that not only include a simulator but also other tools that are needed to ensure certain properties about the system as described in the following.

A simulator is a computer program that attempts to model a system for which a simple analytical solutions cannot be found easily. Simulation allows the study of such a system under a large variety of situations or scenarios.

Simulators play also an essential role in the development of robot controllers. Simulated experiments are usually much faster than real ones and simulated robots do not have hardware failures or battery exhaustion (unless this is an explicitly desired property). Furthermore, simulations do not need to take into account purely technical issues such as calibration of sensors and actuators. Unfortunately, there is no guarantee that the controllers developed in simulation will work as expected in real robots. Hence, a desirable (but not entirely attainable) qualitative property of simulators is the possibility to have a seamless transition between simulation and reality.

There are several approaches to tackle the above problem. A naive solution is trying to model sensors and actuators of the robots as precisely as possible as for making the difference between simulation and reality negligible. Anyway, it is practically unfeasible to perfectly simulate reality (Frigg and Hartmann, 2006), and the more the model becomes more faithful to reality, the more simulation become more complex and hence slower.

On the other hand, physics simulation could be in principle avoided completely. Some sensors such as ground sensors, light sensors and infrared proximity sensors can be implemented by sampling real readings taken from various positions and orientations with respect

to other objects of the environment. The final result is a large numerical table containing the recorded samples that is fairly easy to import and fast to access at run-time. Nevertheless, obtaining such a table is a time-consuming and error-prone activity. Furthermore, different robots with the same sensors are likely to give significantly different readings due to differences in the electronics. Lastly, this technique does not appear applicable to sensors such as digital cameras.

A commonly utilised method to ease to transfer controllers to real robots is adding noise to sensors reading and actuators outputs (Jacobi, 1997). This is a reasonable approach because real sensors and actuators are indeed noisy. Additionally, noise injection softens the natural differences between different physical sensors and actuators. The computational impact of noise addition is negligible but, thanks to it, resulting controllers are more robust and more easily transferable.

The development of a simulator is of critical importance inside the Swarmanoid project. One of the key additional motivation besides the ones highlighted above is the fact that robots are developed during the course of the project and they are not ready before the last periods (at the time of writing, they are still in development). Hence, the only methodology that allows the development of robot controllers before the robots are ready is to use a software simulator. Furthermore, users should also be allowed to seamlessly transfer controllers developed in simulation into the real robots. One way to achieve this is by allowing the same code to be compilable both for the simulator and for the robotic hardware platform without any further intervention. On the other hand, sensors and actuators of the robots need to be simulated as much accurately as possible, thus needing a continuous interaction between people working in hardware and software.

In the following sections we highlight how the design and the code organization of ARGoS and of the entire simulation framework can successfully tackle the above issues. ARGoS is also important for the aim of this report since our proposed architecture (described later in Chapter 4) has been designed within this framework.

2.3.1 The Simulator Architecture

Figure 2.4 depicts the architecture of ARGoS. The key components of the architecture are:

The Swarmanoid Space is the space where all objects live;

The Physics engines are responsible for updating the status (i.e. position, orientation) of the objects they are responsible for;

The Visualization engines are in charge of drawing the scene either under the form of immediate visualization for the user, or in form of text data, or potentially as a high quality movie (at the time of writing not yet implemented);

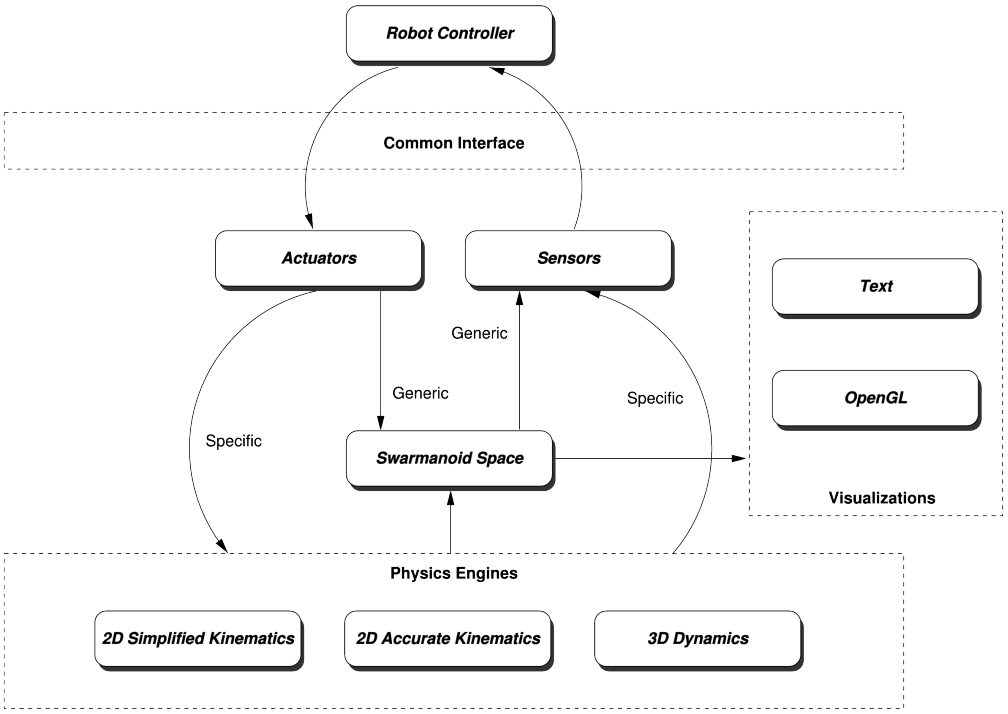


Figure 2.4: Overall architecture of ARGoS, the Swarmanoid Simulator

Sensors and **Actuators** are used by the robots to interact with the environment;

Robot Controllers are programs that associate sensory data to actuator's signals in order to encode the behavior of the robot.

The Swarmanoid simulator has been designed to support the possibility of running different physics engines independently during an experiment. For instance, foot-bots might be simulated in a 2D physics engine, whereas hand-bots and eye-bots could be managed by another, or even other two, separated 3D physics engines. This key feature has been obtained by decoupling the space in which all objects live from the physics engines space. The former, global space goes under the name of *Swarmanoid Space*.

Sensors and actuators are classified in order to provide a coherent structure for their development. Some sensors and actuators rely on physics equations, whereas other sensors, such as the camera, simply rely on positional information to compute their readings. Likewise, some actuators do not need any physical information to perform their actions. Sensors and actuators that are physics dependent and must be reimplemented for each different physics engine are denoted as *specific*, whereas those that do not need such interaction are denoted as *generic*.

The design of ARGoS is highly modular. Each box in Figure 2.4 has been implemented as a plugin. The user can code his own version of each module, and easily inform the system about its existence. Compatibility and interoperability are guaranteed by the interfaces that

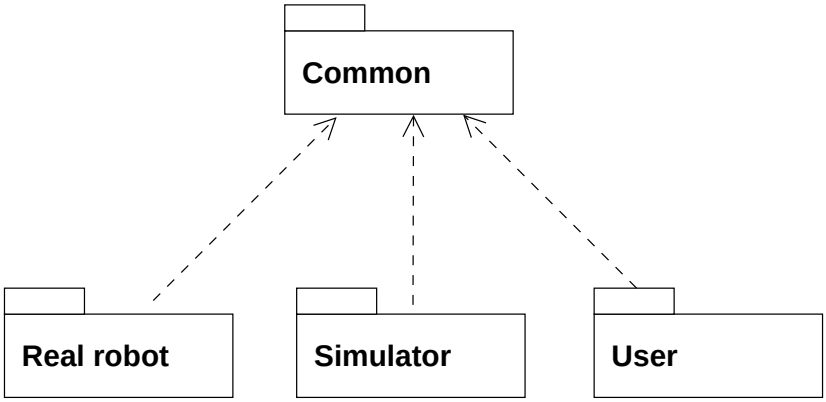


Figure 2.5: The package diagram of the Swarmanoid simulator

will be described in the following section. Particular emphasis will be put on the *common interface*, which left out in this section and is the key component that favors an easy transfer between simulation and real robot.

2.3.2 The Code Organization

Figure 2.5 depicts the software packages in which the code of the entire simulation framework has been divided, and their mutual dependencies. The division in packages has been designed to maximize code reuse while keeping the functionalities logically separated. As we can see, there are four packages: common, simulator, real robot and user.

The **common** package defines an interface to each of the robot's hardware. It is shared across simulated and real robots and enables the user to access the capabilities of a robot in the same way both in simulation and on the real hardware. Additionally, it includes a set of general purpose utilities: we can find string utilities, common definitions, logging facilities, mathematical definitions and functions, and so on. The most important part of this package is the definition of the common control interface, described later;

The **simulator** package contains all the logic and data for the simulator itself. In this package, the common control interface is implemented to provide simulated sensors and actuators. Moreover the Swarmanoid Space, the physics engines and the visualizations as depicted in Figure 2.4 are all included in this software package.

The **real_robot** package contains the toolchain and the wrapper to access the specific robot platform and additional tools. Its purpose is to provide a tool set to compile software for the real robots.

The **user** package allows each user to develop controllers on an individual basis.

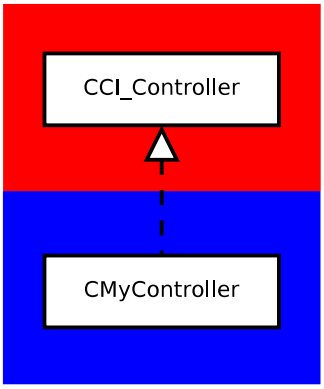


Figure 2.6: The class diagram of a controller which does not exploit the behavioral toolkit

The common package is the package containing the *common interface*. The common interface provides a standard way of accessing the robot hardware APIs. It is written in the C++ programming language. Every controller developed as part of the Swarmanoid project, whether in simulation or on the real robots will access the hardware (real or simulated) through this common interface. Since each robot's hardware is potentially different, also the platform and the set of tools is different as well. Furthermore, heterogeneous interfaces with sensors or actuators might be associated even to a single robot platform, for example because some sensors or actuators might be developed independently as modules. The common interface is an essential component, as it allows controllers to be written only once for both platforms (the one available to the simulator, for example a PC with Linux, and the one available on the robot).

It is out of the scope of this document to provide an in-depth description of all the classes included in the simulator framework (for a more detailed overview refer to Pinciroli (2006)). However, before concluding, we just highlight in Figure 2.6 the class diagram of the control interface while not exploiting the architecture presented in this report. A user made controller is just a straightforward implementation of the common interface class `CCI_Controller`, providing the definition of the functionalities of a controller that are common to the simulation and the real robot. Here, we use a color convention that is common to the one that will be used later, in Chapter 4. According to this convention, a red region denotes a set of classes belonging to the *common* package, whereas blue denotes classes belonging to the *user* package. In Chapter 4, we will show how this architecture changes after the introduction of the behavioral toolkit.

Chapter 3

State of the Art of Robot Software Architectures

The aim of this chapter is provide a brief but structured review of work that has been done in the field of *robot architectures*.

Most of the effort in the area concern single-robot architectures, where much focus has been put into augmenting the robot cognitive capabilities in order to solve complex tasks, often resulting in architectures that are quite complex and of composite nature. On the other hand, some effort has been put in order to provide architectures for multi-robot or swarm systems. Such architectures do not put much stress on the cognitive capabilities of each individual robot, but they rather focus on the orchestration of behaviors in-between several robots.

The rest of the chapter is organized as follows. In Section 3.1 we review the pioneer architectures developed for single-robot systems, while also sketching very briefly what have been their follow-ups. Subsequently, in Section 3.2, we review what we believe are the most representative works done in architectures for multi-robot systems.

3.1 Architectures for Single Robot Systems

The first time researchers felt the need for having a robot architecture dates back in the 1960s with the “Shakey” robot by Nilsson (1969). At that time, all attempts were focused on trying to apply artificial intelligence techniques to robotics in a straightforward way. Those techniques heavily relied on the presence on a internal model inside the robot. To deal with this, the architecture that was wide spread at that time was called SPA (Sense Plan Act). It was composed of three subsystems: the *sensing* sub-system was translating sensory data into the internal robot model; the *planning* sub-system took the internal world model and a goal and generated a plan achieving the goal; the *execution* sub-system took the plan and

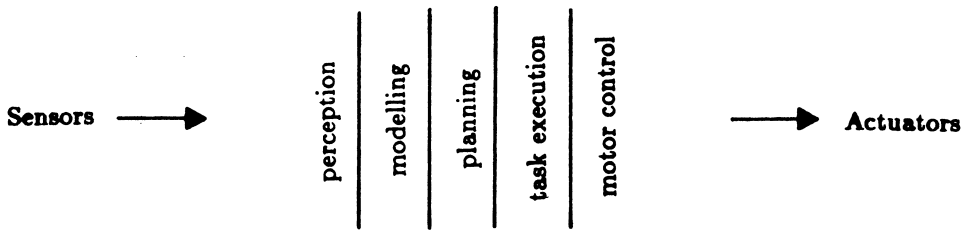


Figure 3.1: Schema of the mobile robot control system architecture used earlier than the introduction of the subsumption architecture

converted it into actions for the robot.

The above control paradigm was widely used in robotics for many years. In the early 1980s, researchers realized that SPA had many problems: first, the planning phase was very slow or unfeasible for most of the real world problems; second, acting in the real world by only using an internal model without relying to the sensors was counterproductive especially in changing environments; third, no direct mapping exists between sensors to an internal world, due also to noise in the sensors. Some new paradigms were developed in order to face the issue, amongst which the most influential was, without doubts, the seminal work of Brooks (1986).

The aim of this section is to briefly introduce the subsumption architecture (in Section 3.1.1) together with the general trend that was followed in the following years until nowadays (in Section 3.1.2).

3.1.1 The Subsumption Architecture

The subsumption is an architecture which is composed of layers, each of which is composed of asynchronous modules which communicate using channels with minimal requirements. Each module is an instance of a very simple finite state machine (Minsky, 1967), which interconnects sensors to actuators directly. These modules were renamed as *behaviors*. The presence of multiple layers or level of competence induces also an implicit arbitration mechanism, as we will see later.

The SPA architecture widely used before the subsumption can be exemplified in the schema of Figure 3.1[1]. On the other hand, Brooks' subsumption consists in a vertical schema as the one depicted in Figure 3.2. As we can see, the functional decomposition in modules is almost the same in both architectures. What changes is that, in the traditional approach, the problem is decomposed according to the internal workings of the solution, that is intermediate modules like planning are not directly connected to sensors and actuators. On the other hand, in the subsumption each module (including planning) always faces

[1] All figures in this section were taken from Brooks (1986)

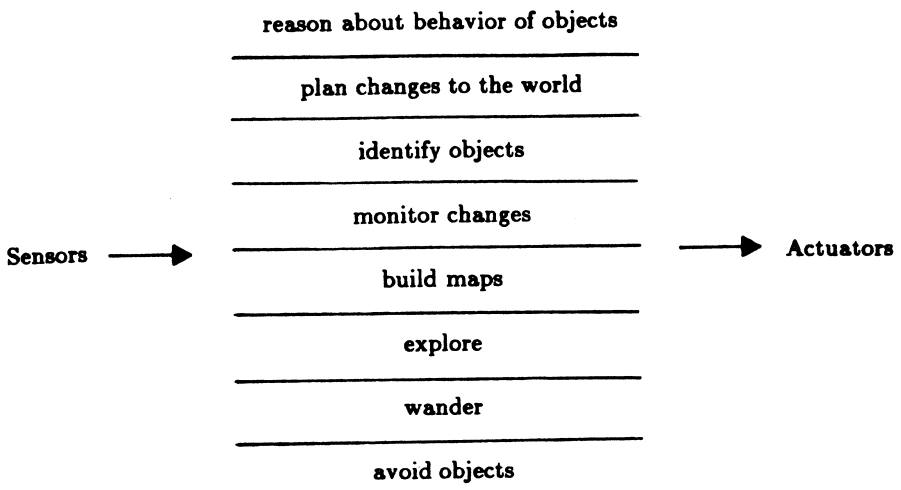


Figure 3.2: Schema of the mobile robot control system architecture used in the subsumption architecture

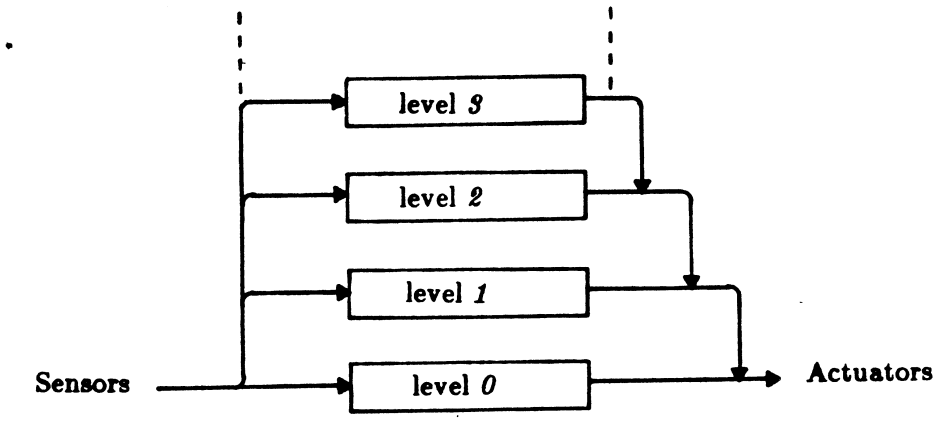


Figure 3.3: The role of levels of competence in the subsumption architecture

the outside world, leading to a representation that is designed on the basis of the desired external manifestations of the robot control system.

The basic concept of the subsumption architecture is the one of the *level of competence*. Figure 3.3 shows how this concept is put into use. A level of competence collects a set of behaviors that pertains to some class. Classes are defined according to some informal specification. The higher the level of competence, the more specific the class of behaviors should be, i.e. the more constraints it should put on the behaviors. This organization also induces a coordination mechanism amongst different levels: behaviors at an higher level of competence can inhibit signals coming from lower level behaviors (Figure 3.4).

Consider the example proposed by Brooks (1986). A level zero level of competence can

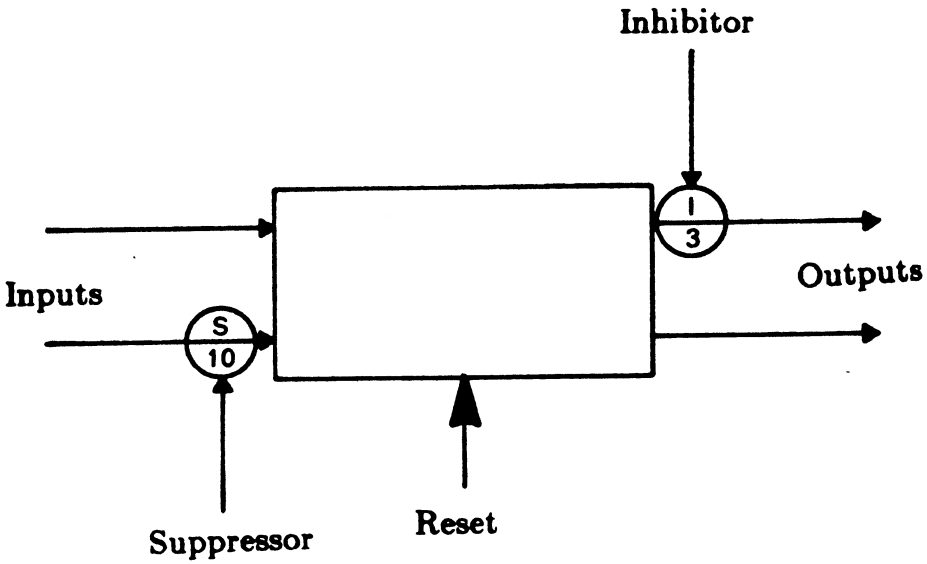


Figure 3.4: Schema representing one module in the subsumption architecture

be represented by the behaviors depicted in Figure 3.5. We assume the designer has provided a set of behaviors that are able to let a robot move away from obstacles in a cluttered environment. Imagining an environment without obstacles, the result of level 0 behaviors has the robot probably ending up in the middle of it. To get a more interesting behavior we can add a further level (Figure 3.6). Here, we can imagine having a wander module continuously generating random headings, and an avoid module which will perform a more clever obstacle avoidance by combining some of the output of the earlier layer with the output of the wander module. In this example, when the presence of an object is detected, the output of the wander module will be overwritten in order to produce a trajectory able to avoid the object. If the object is not present, the wander behavior will have control of the trajectory generation, producing a behavior that can explore the environment aimlessly. As a result, the two combined layers produce a robot behavior that is able to aimlessly explore the environment while avoiding obstacles. On top of these two, other layers can be engineered in order to face more restrictive specifications, for example in order to construct a map of the environment.

In his original work, Brooks (1990) proposed also a Behavioral Language, which was mainly inspired by the LISP[2] functional programming language. More recently, implementations in more modern imperative programming languages were proposed, for example by G. Butler (2001).

One of the main advantages of the subsumption architecture is that behaviors can be

[2]McCarthy, John (1979-02-12). "The implementation of Lisp". History of Lisp. Stanford University. http://www-formal.stanford.edu/jmc/history/lisp/node3.html.

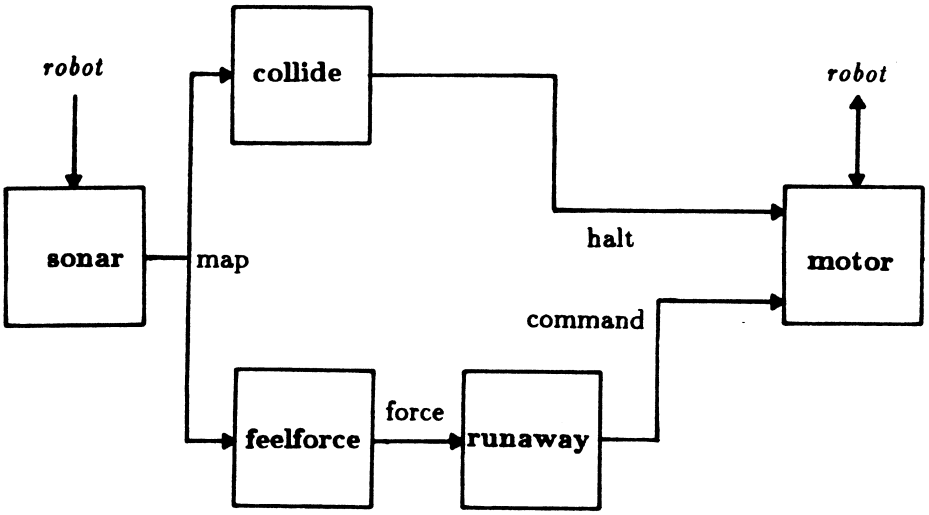


Figure 3.5: An example of subsumption architecture with level of competence 0 only

constructed, tested and debugged independently. The introduction of a new behavior or of a new level of competence does not require the designer to rethink all the modules that she has produced so far together with their interconnection, as it was the case for the traditional design. Furthermore, subsumption proved to be successful when compared to SPA robots: the latter were slow and ponderous, whereas the former produced robots that were fast and reactive. A changing environment was not an issue for them since they constantly sensed the world and reacted to it. Finally, the subsumption architecture served as an inspiration for several other follow-ups. These subsequent contributions will be very briefly outlined in the coming section.

3.1.2 Other Single-Robot Architectures

The aim of this section is to briefly sketch some (and hopefully the most important) of the remaining architectures for single robot systems. This small review does not have the ambition to be comprehensive. For a more complete review, refer to specialized books such as Siciliano and Khatib (2008), Arkin (1998), Kortenkamp et al. (1998), Murphy (2000) and Siegwart and Nourbakhsh (2004).

Motor Schemas Developed by Arkin (1989), it was the natural follow-up of the subsumption. It is a biologically inspired approach, where motor and perceptual schemas are dynamically connected to each other. Each schema generates response vectors based on outputs of the perceptual schemas, which are then combined together in a way similar to potential fields.

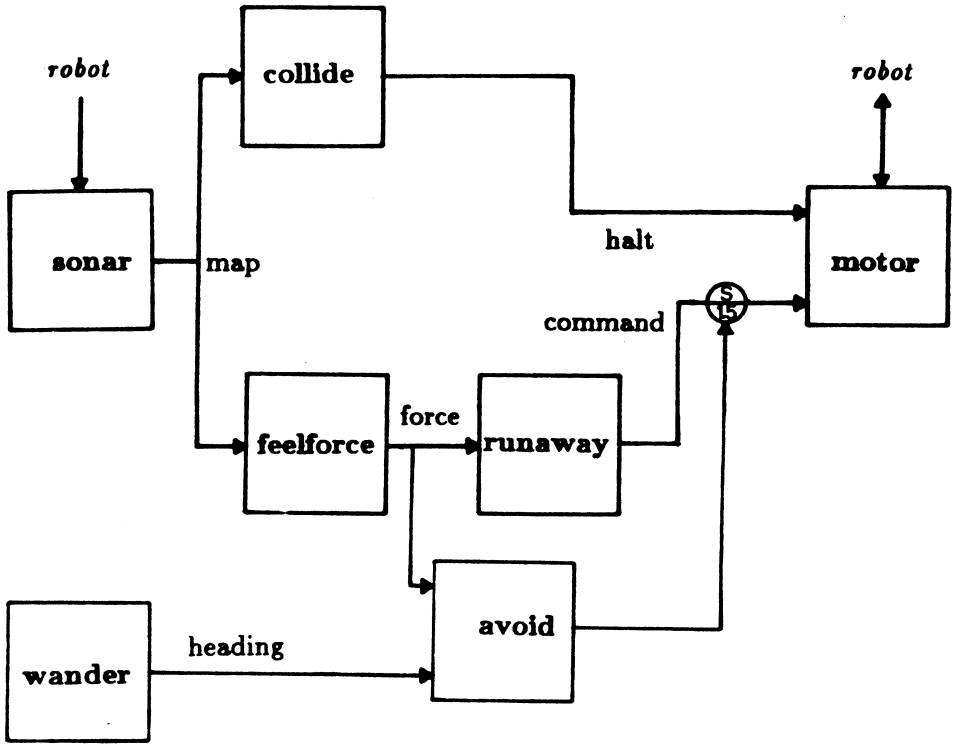


Figure 3.6: An example of subsumption architecture with level of competence 0 integrated with level 1

AuRA As well developed by Arkin (1998), it enhances motor schemas by adding a navigation planner and a plan sequencer based on finite-state acceptors (FSAs). It was one of the first *hybrid* architectures.

MITRE Introduced by Bonasso (1991), it was the first 3T (three tier) architecture, which consists in three interacting layers, namely planning (high level), sequencing or execution (intermediate level) and real-time control (low level).

ATLANTIS Introduced by Gat (1992), it is also a 3T architecture. The central concept to the ATLANTIS architecture is that of "activities at varying time scales". At short time scales the world is both dynamic and very precise, as the time scale grows the world becomes more static and more abstract. As a result, each layer addresses a different time scale.

Intelligent Control Developed by Saridis (1995), it is also a 3T architecture. Here, the control layer is implemented through low level servo systems, the sequencing layer through Petri net transducers, and the planning level through Boltzmann neural networks.

LAAS Another 3T architecture developed by Alami et al. (1998), it consists in a lower layer

composed of parameterizable modules written in a module generator (GenoM), a sequencer layer that is very simple and purely reactive and simply maps the highest level to the lowest one by generating formally verifiable decision networks, and an higher layer consisting in a planner implemented using a temporal planner and a reasoning system.

Two-layered Alternatively to 3T, two-layered architectures were also developed. Examples are CLARAty (Volpe et al., 2001), CLEaR (Estlin et al., 2005), CIRCA (Musliner et al., 1995) and ORCCAD (Espiau et al., 1995).

As we just saw, most of the architectures for single-robot systems consist of two or three tiers which include a deliberative component or planning layer. It is clear that, in order to tackle task complexity with single-robot systems, a deliberative component is necessary. Such component should be able to arbitrate the different low level components in an optimal way On the other hand, by using a multiple robot approach we avoid needing a deliberative component since we can scale in complexity by increasing the number of robots involved in the system. This has the clear advantage of not requiring much complexity both in terms of hardware and of software components on each individual robot. In the next section, we review the most prominent robot architectures that were expressly thought and designed for multi-robot systems.

3.2 Architectures for Multiple Robots and for Swarm Robotics

Designing an architecture for multi robot systems that would scale as the complexity of the task and the number of robots increases is not trivial. As a first naive guess, one can attempt to take an existing architecture for single robot systems and apply it straightforwardly to multiple robots. This is indeed what was attempted in the early 90s. The subsumption architecture is a good candidate for this type of experiment since it is very simple and consists in a single tier and, intuitively, this could allow the scalability with respect to the number of robots.

Mataric (1992) studied if a good social behavior would emerge from a group of robots implementing very simple, subsumption based controllers. Her very simple experimental setup involved a task where a group of 20 identical robots were given an identical location as a goal, which in turn would be placed on a different region of the environment separated from the start location by a narrow door where only one robot could pass. She compared three types of controllers:

Ignorant Coexistence Each robot was implementing two behaviors only: *go to goal* and *obstacle avoidance*. This type of controller threated other robots as mere obstacles. This type of controller lead to a very slow achievement of the task. Furthermore, the

more the number of robots, the worse was the traffic jam generated and the slower the performances.

Informed Coexistence This controller used the same behaviors of the ignorant coexistence controller plus a third one: *avoid robot*. If the robot detected another fellow robot, it would stop and wait for a time p. The results were much better in this case and were comparable to having a fixed robot running back and forth the door 20 times.

Intelligent Coexistence In this controller, the "social behavior" used in Informed Coexistence was replaced by another one: when a robot detected another robot, it would move away from it but, at the same time, it would try to move in the same direction as the majority of the other robots (this was achieved by broadcasting a common heading direction through a radio channel). This controller surprisingly resulted in one line of robots being generated and going towards the door in one shot. As a result, a much better behavior (yielding to much less traffic jam) emerged from the interaction of very simple behaviors.

Although results were encouraging, some key aspects of collective and swarm robotics such as robustness, fault tolerance and adaptivity were not considered. As a matter of facts, in the experiment the team of robot could not actively help out failed colleagues or change the task dynamically. The first notable attempt to enforce an architecture for collective robotics that would deal with these key aspects was developed by Lynne Parker and is described in the following.

3.2.1 The ALLIANCE Architecture

ALLIANCE (Parker, 1998) can be considered an outgrowth of subsumption. The problem that this architecture tackles is the achievement of fault tolerant, robust and adaptive task allocation in a team of robots. The central idea of this architecture is that robots in a team can sense the "progress" of other team members in achieving a certain task, additionally to their own progress. When they get "frustrated" with their own progress, they should start doing something else (change task). Likewise, if a robot is free and perceives the non-achievement of some task by another robot, it should take over and try to complete it.

The central concept of ALLIANCE is the one of *motivation*. The motivation is in turns linked to two other concepts: *impatience* and *acquiescence*. The impatience measures the degree of frustration of a robot regarding other robot's behavior's performance with respect to some task T_i. Likewise, acquiescence measures the degree of frustration of a robot regarding its own behavior's performance on task T_i.

The ALLIANCE architecture can be formally described as follows.

Preliminary Definitions

Suppose having a team of of n heterogeneous robots $\mathbb{R} = \{R_1, R_2, \ldots, R_n\}$, a set of m tasks $\mathbb{T} = \{T_1, T_2, \ldots T_m\}$ and a set of an arbitrary number of behaviors for each robot i, $\mathbb{A}_i = \{A_{i1}, A_{i2}, \ldots\}$. A set of n functions $\{h_1(A_{1k}), h_2(A_{2k}), \ldots, h_n(a_{nk})\}$ is used to determine the task in $\mathbb{T}$ associated to robot $i \in \{1, \ldots, n\}$ when it activates behavior A_{ik}. Each behavior can be activated according to sensory feedback:

$$sensory\_feedback_{ij}(t) = \begin{cases} 1 & \text{if, for } R_i \text{ behaviour } A_{ij} \text{ is applicable at time } t, \\ & \text{and} \\ 0 & \text{otherwise.} \end{cases}$$

The communication can be, in turn, modeled as follows:

$$comm\_received(i, k, j, t_1, t_2) = \begin{cases} 1 & \text{if } R_i \text{ has received a message from } R_k \text{ concern-} \\ & \text{ing task } h_i(a_{ij}) \text{ in time interval } [t_1, t_2] \\ 0 & \text{otherwise.} \end{cases}$$

Futhermore, in ALLIANCE, a robot can perform only one task at a time. Hence, a suppressor signal is needed:

$$activity\_suppression_{ij}(t) = \begin{cases} 0 & \text{if another behaviour } A_{ik} \text{ is active, } k \neq j, \text{ on} \\ & \text{robot } R_i \text{ at time } t, \text{ and} \\ 1 & \text{otherwise.} \end{cases}$$

Other parameters need to be introduced before defining impatience and acquiescence.

- $\phi_{ij}(k, t)$ denotes the amount of time robot R_i allows R_k to affect the motivation of behavior A_{ij}

- $\delta\_slow_{ij}(k, t)$ and $\delta\_fast_{ij}(t)$ denote the rates of impatience of robot R_i for behavior A_{ij}

- $\psi_{ij}(t)$ gives the time that robot R_i wants to perform a task before yielding to another robot.

- $\lambda_{ij}(t)$ gives the time robot R_i wants to mantain the task before giving up and possibly trying another one.

Impatience and Acquiescence

The impatience rate can be defined as:

$$impatience_{ij}(t) = \begin{cases} min_k(\delta\_slow_{ij}(k,t)) & \text{if } comm\_received(i,k,j,t-\tau_i,t) = 1 \text{ and} \\ & comm\_received(i,k,j,0,t-\phi_{ij}(k,t)) = 0 \text{ and} \\ \delta\_fast_{ij}(t) & \text{otherwise.} \end{cases}$$

Intuitively, the impatience rate is set to the minimum $\delta\_slow_{ij}(k,t)$ if robot R_i has received a communication signal from R_k indicating that the latter has been performing task $h_i(a_{ij})$ for the previous τ_i time units but it has not lasted longer than $\phi_{ij}(t)$ time units. Otherwise, the impatience rate is set to $\delta\_fast_{ij}(t)$.

The motivation of a robot to perform a task is reset to 0 the first time it hears about another robot performing the same task, i.e. if robot R_i received its first message from R_k indicating it is performing task $h_i(a_{ij})$. We can use βt to denote the last time a robot communicated with the others. Hence, formally:

$$impatience\_reset_{ij}(t) = \begin{cases} 0 & \text{if } \exists k : comm\_received(i,k,j,t-\beta t,t) = 1 \text{ and} \\ & comm\_received(i,k,j,0,t-\beta t) = 0 \text{ and} \\ 1 & \text{otherwise.} \end{cases}$$

The acquiescence is defined as follows:

$$acquiescence_{ij}(t) = \begin{cases} 0 & \text{if } (A_{ij} \text{ of } R_i \text{ has been active for more than } \psi_{ij}(t) \text{ time units at time } t \text{ and } \exists k : \\ & comm\_received(i,k,j,t-\tau_i,t) = 1 \text{) or } (A_{ij} \text{ has been active for more than } \lambda_{ij}(t) \text{ time units at time } t), \text{ and} \\ 1 & \text{otherwise.} \end{cases}$$

Motivation

Finally, the motivation m_{ij} of robot R_i to perform task T_j can be evaluated as follows:

$$m_{ij}(0) = 0$$

$$\begin{aligned} m_{ij}(t) = \; & [m_{ij}(t-1) + impatience_{ij}(t)] \\ & \times sensory\_feedback_{ij}(t) \\ & \times activity\_suppression_{ij}(t) \\ & \times impatience\_reset_{ij}(t) \\ & \times acquiescence_{ij}(t). \end{aligned}$$

The robot can then use a parameter θ as a motivation threshold: if $m_{ij} > \theta$, the corresponding behavior is activated. All robots broadcast their current activities as signals at a given rate ρ_i.

L-ALLIANCE

Parker (1997) also developed a mechanism that allowed the adaptation of parameters used in ALLIANCE: $\delta\_fast_{ij}(t)$, $\delta\_slow_{ij}(k,t)$ and $\psi_{ij}(t)$. Such an augmented framework was called L-ALLIANCE. The performance metric used was $task\_time_i(k,j,t)$, which denotes the average time over last μ trials of R_j's performance of task $h_i(a_{ij})$ augmented by the standard deviation of these μ trials as measured by robot R_i.

The parameters are updated as follows:

$$\phi_{ij}(k,t) = task\_time_i(i,j,t)$$

$$\delta\_slow_{ij}(k,t) = \frac{\theta}{\phi_{ij}(k,t)}$$

$$min\_delay = \text{minimumalloweddelay}$$

$$max\_delay = \text{maximumalloweddelay}$$

$$high = \max_{k,j} task\_time_i(k,j,t)$$

$$low = \min_{k,j} task\_time_i(k,j,t)$$

$$scale\_factor = \frac{max\_delay - min\_delay}{high - low}$$

$$z_{case_1} = \frac{\theta}{min\_delay - (task\_time_i(k,j,t) - low) \cdot scale\_factor}$$

$$z_{case_2} = \frac{\theta}{min\_delay + (task\_time_i(k,j,t) - low) \cdot scale\_factor}$$

$$\delta\_fast_{ij} = \begin{cases} z_{case_1} & \text{if the robot expects to perform the task better than any other team member and no robot is currently performing it, and} \\ z_{case_2} & \text{otherwise.} \end{cases}$$

$$\psi_{ij}(t) = task\_time_i(i,j,t)$$

The reader that is interested to go more into the details can refer to the original papers Parker (1998, 1997).

Advantages and Disadvantages

The ALLIANCE architecture has the advantage of being fully distributed and intrinsically fault tolerant and cooperative. Robots adapt their behavior automatically, even when a centralized knowledge is absent. It does not assume nor require availability of a communication medium, neither does it require the agents to be fully reliable. In fact, it was designed to work also in case of such problems arise. The absence of communication is seen as a worst-case scenario.

However, ALLIANCE is based on the following assumptions:

1. Robots can detect the effect of their own actions;
2. Robots can detect the effects of other robots actions, by any available means including broadcast communication;
3. Robots act cooperatively (they do not “lie”);
4. The communication medium is not guaranteed to be available;
5. Sensors and actuators are not assumed to be perfect;
6. Any robot can fail;
7. If a robot fails, we do not assume its communication to the other robots;
8. A centralized store of world knowledge is not available.

By analyzing these assumptions, we can see that the most restricting one is assumption 2. Even if a communication medium is not strictly required explicitly, assumption 2 implicitly enforces this requirement since there is no other trivial way for a robot to know other robots' activities without explicit communication. Not only that, but such communication needs also to happen in broadcast. As a result, the ALLIANCE architecture is not directly applicable to swarm robotics, where only local knowledge is assumed to be available at most.

3.2.2 The ASyMTRe Architecture

According to a taxonomy developed by Gerkey and Mataric (2004), ALLIANCE is an architecture which can tackle multi-task (MT) single-robot (SR) type of task allocation problems. These are also called *weakly-cooperative* problems, i.e. problems composed of multiple tasks where we assume that each individual task can be solved by one robot independently.

In Parker and Tang (2006), another architecture was proposed to tackle single-task (ST) multi-robot (MR) type of problems. Here, they assume that even a single task cannot be solved by a robot alone, but low level-type cooperation is needed in order to tackle it. Such

problems are also referred to as strongly cooperative. In the following, we provide an overall qualitative description of ASyMTRe. The reader that wants to go more in the details can refer to Parker and Tang (2006); Tang and Parker (2005a,b).

ASyMTRe (which stands for Automated Synthesis of Multi-robot Task solutions through software Reconfiguration) can be considered a two-tier architecture. At the lowest level, schema based are used to implement low level behaviors (Arkin, 1989). At an high level, the ASyMTRe algorithm (which stands for "Automated Synthesis of Multi-robot Task solutions through software Reconfiguration" is used to automatically reconfigure robot schemas in order to address the task at hand. Inspired from and extending Arkin (1989), they use the following as building blocks for the low level tier:

PS Perceptual Schemas are used to process input from environmental sensors to provide information to motor schemas;

MS Motor Schemas generate output control vectors representing how the robot should react in response to the perceived stimuli;

CS Communication Schemas, which are new in ASyMTRe, transfer information between various schemas distributed across multiple robots.

All those schemas are alread pre-programmed into the robot at design time. What is lacking at design time is the connection amongts schemas. Such interconnection is performed dynamically and autonomously at run-time, using the ASyMTRe algorithm.

The ASyMTRe algorithm is, in the original formulation (Tang and Parker, 2005a), a centralized reasoner, which generates solutions with increasing quality as more time is available for the process (such type of algorithms belong to the category of *anytime algorithms*). The algorithm first starts from an original configuration space (OCS), which contains all the possible schemas configuration combinations. The complexity of the space is exponentially large in the number of robots. In order to reduce the complexity, the algorithms subsequently generates a potential configuration space (PCS) by dividing all solutions into classes of equivalence and considering only one solution per class. The algorithm then performs search in such a reduced space, ensuring that at anytime it can be polled for the solution which corresponds to the highest maximum total utility at that time (which is a measure of global performances).

In a more recent work, Tang and Parker (2005b) proposed a distributed version of ASyMTRe, denoted as ASyMTRe-D. They setup a Contract Net Protocol that implements a distributed negotiation process that can be put in place of the centralized mechanism. The approach is suitable also for applications where hardware failures are common. However, a trade-off between solution quality and robustness, or alternatively between full centralization and full distributedness, must be taken into account.

Although very promising, ASyMTRe and ASyMTRe-D are both not ready to be applied in swarm robotics. The negotiation protocol used in ASyMTRe-D does rely on the possibility of performing broadcast of messages, since the information that might be needed for a robot to self-configure itself might be located in any or multiple robot in the team.

In the next section, we analyse one of the most commonly used architectures in swarm robotics. Although it is not very often referred to as an architecture, it is indeed a common paradigm that is used and often also analysed analytically.

3.2.3 Probabilistic Swarm Robotics Architecture and Modeling

As already stated in Chapter 1, swarm robotics is a very challenging domain. One of the most important desirable property in such a domain is scalability, i.e. a group or swarm of robots should be able to solve a task effectively or more effectively as the number of robots increases. Solutions that require either global knowledge about the environment or explicit all to all communication should hence be avoided.

So far, there is no complete architecture in swarm robotics that exhibits such features. Nevertheless, there is somehow a common recipe that is very often followed when writing robot behaviors. According to such approach, the individual robot behavior consists in a simple finite state machine (FSM). Furthermore, some of the transition between states in such a FSM might be governed by the so called response threshold model (Granovetter, 1978). According to the model, the probability of transitioning from one state to the other has the following form:

$$P = \frac{N}{N + k^{\beta}}$$

where N represents a threshold, β a sensitivity parameter and k represents some stimulus that might be obtained from the environment and/or from the social interaction with other robots. Such model was introduced in swarm robotics by Bonabeau et al. (1997) and was then followed in many other works (Nouyan et al., 2008; Soysal and Sahin, 2005; Liu et al., 2007). It has demonstrated very good scalability in presence of multiple robots in non-trivial tasks, despite its simplicity.

In a swarm of robots, when an individual robot implements a type of behavior such as the one we just described, the knowledge about the global task is nowhere explicitly encoded in the behavior. Nevertheless, if the individual behaviors are designed *properly*, a global collective behavior to solve the task *emerges*. Unfortunately, so far there is no recipes that tells, given the global task we want to solve, how to design the individual behaviors “properly”.

Nevertheless, some works have been devoted into modeling swarm systems. Such works starts from a controller for an individual robot that has already been designed. Then, a macroscopic model is constructed by applying a direct mapping. Such methodology is often referred as bottom-up (for a comparison between top-down and bottom-up methodology refer

to Crespi et al. (2008)).

A good review on modeling swarm robotics behaviors can be found in Lerman et al. (2005). Such a modeling procedure can be summarized in the following phases:

1. Starting from an individual level FSM, a collective level FSM which is functionally identical can be derived. The difference between the individual and the swarm level FSM is that in the swarm level FSM each state can be interpreted as the number of robots in that state;

2. The macroscopic FSM can directly be translated into Rate Equations (Lerman and Galstyan, 2002). Each state in the FSM becomes a dynamic variable $N_n(t)$, with its own Rate Equation;

3. Every transition can be translated into a term of the above equation: a positive term $+W$ for each ingoing arrow and a negative term $-W$ for each outgoing arrow.

Phase 3 is in general the most challenging one. As we said, transitions are in general governed by the amount of stimuli perceived from the environment and from the social interaction with the other robots. Assuming that robots and stimuli are uniformly distributed, the transition terms in 3 can be approximated by $W \approx M$ where M is the amount of environmental stimulus encountered. These quantities can, in turn, be derived from known principles, measured in simulations or with experiments with real robots. Alternatively, they can be left as parameters of the model and obtained later by fitting the model to data.

Such macroscopic models are very useful in swarm robotics. First, they can be used in place of simulation since, comparatively, it is a much more scalable approach as the number of robots increase arbitrarily. Second, they can help to provide a better understanding on how environmental and social stimuli impact on the mapping between individual behaviors and the global behavior. Finally, those insights can be, we believe, of critical importance for the design of a more complete architecture for swarm robotics, which is still a challenging direction for future works.

Chapter 4

The Behavioral Toolkit

Chapter 3 reviewed several state of the art robotics architectures. In this chapter we will introduce our own architecture, i.e. the behavioral toolkit. We will describe how the proposed architecture represents a general way of programming modular, behavior based controllers, of which the subsumption architecture represents a special case.

The chapter is organized as follows: we will first describe the motivations that justify the development of such a toolkit; we then present the main idea and concepts, such as the one of *Robot State*; finally, we show how the behavioral toolkit has been implemented in the ARGoS simulator.

4.1 Motivation

Development of software in the robotics framework is a very challenging task. In other areas of software development, software engineering has contributed to establish the initial design principles, patterns and practices in order to ensure some properties of the developed system such as separation of concerns, modularity, abstraction, code reuse, incremental development, etc . . . The traditional software industry has without any doubts benefited since the introduction of such principles. On the other hand, such contributions have encountered many difficulties in their applications in peculiar domains such as robotics. Key reasons of this might include:

1. Robot controllers belong to a special category of software systems called real-time systems, i.e. systems that are subject to a “real-time constraint” (in this case the control cycle length). Hence, traditional principles and techniques developed for traditional software cannot be applied here as they are;

2. The level of complexity of the solution required to solve certain tasks in robotics is often too complex to motivate the development or the use of general purpose solutions;

3. Robot hardware is in continuous evolution and each lab produces its own solution and/or integration of existing solutions;

4. As a result of 3., software middle-ware is often unstable or inexistent.

Notwithstandingly, an attempt to introduce software engineering principles inside robotics would, we believe, benefit the community in the medium/long term.

We developed the behavioral toolkit in the hope of providing some of the principles of software engineering inside the development process of controllers for the Swarmanoid project. So far, controllers for the Swarmanoid project have been developed following a monolithic approach able to produce ad-hoc solutions for specific tasks. Such approach has been proved to be effective in case the complexity of the task to solve is kept fixed. However, such an approach is likely to fail in case the complexity of the task increases arbitrarily. Furthermore, code re-usability is in general not ensured since it is not clear how to identify self-contained modules or to re-use them in different controllers. As a consequence, it is not easy to share code between different developers.

The behavioral toolkit architecture has been thought to overcome the above limitations. In fact, it has been developed by keeping in mind central concepts such us modularity and code re-usability. In the following sections, after describing the main idea behind the architecture, we will also describe what implementation in ARGoS together with few examples that show the difference in how to write controllers between with and without the behavioral toolkit. More complex examples are provided in Chapter 5.

4.2 The Main Idea

Motivated by what already said in the previous section, we require the following key properties from our architecture:

- The controller code should be splittable in code modules, henceforth called *behaviors*;
- Behaviors should be small, maintainable and readable, in order to guarantee an easier debug and test both independently and together with other modules;
- Interaction between different behaviors should happen in a black box fashion, in order to guarantee code reuse;
- Code should also be well organized and well structured inside an individual behavior.

The key component required to ensure the decomposability of controllers into behaviors is a standard interface between a behavior and the outside world, to ensure the interaction with other behaviors. In our behavioral toolkit, such an interface is instantiated as what we call

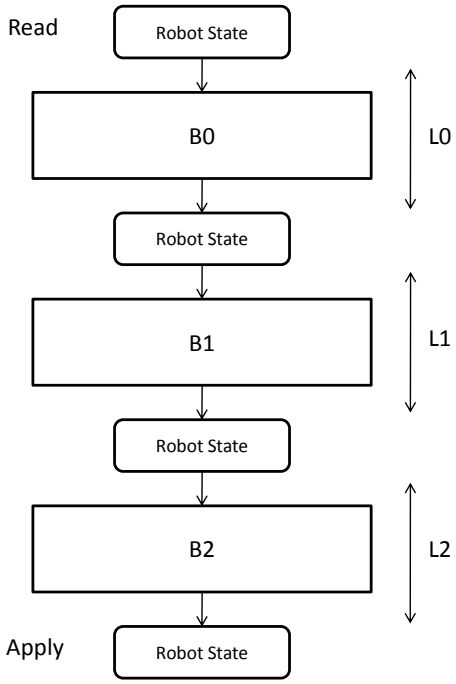


Figure 4.1: The behavioral toolkit architecture instantiated in the case of serial behaviors only. B0, B1 and B2 denote behaviors, whereas L0, L1 and L2 denote levels of competence

a *robot state*. A robot state is a wrapper around the complete sensor and actuator space of the robot. At each time-step, the robot state contains a photography of the values of all the sensors at that time-step. This information is shared and is kept fixed amongst all behavior acting at that time-step. Furthermore, it contains also a copy of all the actuators' statuses. Such an information can be modified by each behavior inside a single time-step. The order with which behaviors write into the actuators determine the final, overall outcome of the controller.

To better structure the explanation of the architecture, we split it into two different sections. In Section 4.2.1 we will depict how it is possible to combine the action of two or more behaviors into another, higher-level behavior, under the assumption that they are executed in sequential order. Next, in Section 4.2.2, we will examine the more general case of behaviors acting in parallel. Finally, Section 4.3 is devoted to more practical examples on how the architecture is implemented and how to use it in the ARGoS simulator.

4.2.1 Sequential Behaviors

Figure 4.1 depicts a schematic that represents the behavioral toolkit implementing a behavior that combines three other behaviors executed in sequential order. It represents the control

execution of a single time-step of the robot control. As mentioned already, the most central concept of our architecture is the robot state. At the beginning of the time-step, such a state is *read*. This simply means that at that point, all sensors are polled and all values are written within the state. Immediately afterwards, the state starts flowing inside the three behaviors. In this example, behaviors have been organised in level of competence, organisation that resembles the one of the subsumption architecture already introduced in Section 3.1.1. The first behavior, denoted as $B0$, with the lowest level of competence (i.e. $L0$) can read the sensorial information through the robot state and decide or vote, after some computations, on some values to put inside the actuators. Subsequently, the state is injected into the following behavior $B1$, in this example having level of competence $L1$. This behavior can, in turn, check the sensorial information (that is still the same read by the previous behavior), but it can also investigate, during its computations, on the status of $B0$ before deciding whether to keep some or all the output decided by $B0$ or to discard/overwrite some or all of them. The robot state then exits $B1$ and enters $B2$ which belongs to level $L2$, where a similar process takes place. After all behaviors have been executed, the final outcome is actually written into the actuators through a *write*. The time-step has thereafter elapsed and the execution flows continues to the next time-step.

In order to better understand how the architecture works, consider the following very simple example. The example consist in a controller for the foot-bot (or for any wheeled mobile robot) composed only of two behaviors, $B0$ and $B1$, organised in sequence. Furthermore, let $B0$ be the behavior *Random Walk* and $B1$ the behavior *Obstacle Avoidance*. At the beginning of the time-step, the robot state is read and injected into *Random Walk*. This particular behavior has no need to obtain information about the sensors value, but it only applies a random rotation for each wheel. This action (random rotation of the wheel) is inserted in the robot state and not directly applied to the real actuators. Subsequently, the robot state is injected into *Obstacle Avoidance*. This time, the behavior needs to check the sensors' values. In particular, it needs to check the values of the proximity sensors to detect whether there is an obstacle or not. If the obstacle is detected, then the behavior will write a value of the robot's wheels into the robot state that will prevent the collision from happening, for example forcing the robot to move in the opposite direction of the obstacle. If the readings, instead, say that the obstacle is not present, than *Obstacle Avoidance* will not write anything into the robot state. As a consequence, the final write will propagate a different information to the real actuator according to the status of *Obstacle Avoidance*: in case the obstacle is not present, the propagated value will be whatever random values *Random Walk* wrote earlier; in case the obstacle is not present, values set by *Obstacle Avoidance* will propagate.

In this example we already see how modularity and incremental development are guaranteed. A *Random Walk* behavior can potentially be implemented, tested and debugged alone. A *Obstacle Avoidance* can then be written on top of it, which means that this latter

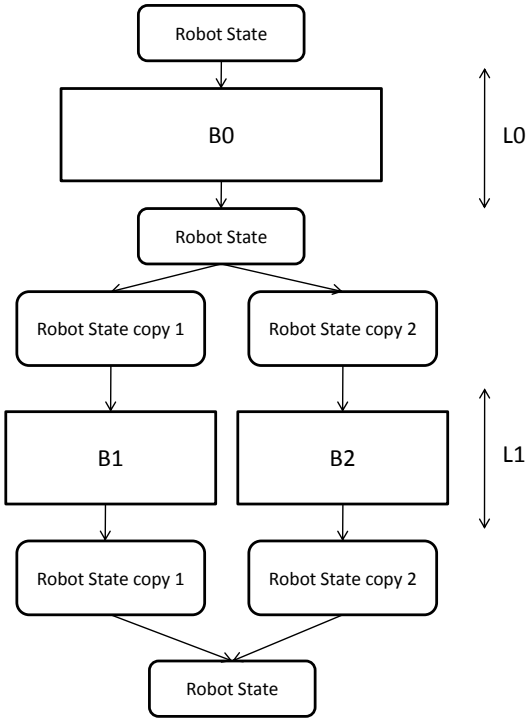


Figure 4.2: The behavioral toolkit architecture instantiated in the more general case of parallel behaviors. B0, B1 and B2 denote behaviors, whereas L0, L1 and L2 denote levels of competence

behavior can "use" *Random Walk* in a black-box fashion.

4.2.2 Parallel Behaviors

Figure 4.2 depicts a schematic that represents the behavioral toolkit implementing a behavior that combines several other behaviors executed in sequence or in parallel. This represents the most general usage of the behavioral toolkit.

As in the previous example, the diagram represents the control execution of a single time-step of the robot control. Again, at the beginning of the time-step, the robot state is *read*. In this example, three behaviors, i.e. $B0$, $B1$ and $B2$, are present. However, they are organised in only two level of competence, $L0$ and $L1$. This means that there are two behaviors that belong to the same level of competence (in this example $B1$ and $B2$). If two or more behaviors belong to the same level of competence, it means than none of the two has a higher priority than the other one. In other words, none of the two behaviors can decide to inhibit or overwrite the other one. So, who is responsible for deciding how to take into account the contributions of two concurrent behaviors? In our example, the overall behavior represented by the entire schematic is the responsible for it.

In this example, $B0$ is executed at first and then, sequentially afterwards, $B1$ and $B2$ are executed in parallel. However, we cannot allow the "main" robot state to be injected in any of the two behaviors, since none of the two is allowed to modify the actuators' variables. To solve this issue, a simple but effective copy functionality has been provided. Before entering any of the two behaviors, two copies of the robot state are created. Any modification to any of these copies will not affect the main robot state and hence will not influence the final values of the actuators' variables. Each of the two robot state copies is injected into the corresponding behavior. After they perform their computations, two potentially different robot states are returned. It is then the duty of another behavior, which can be considered a combiner behavior, to perform the computation of the final robot state, based on the individual contributions of $B1$ and $B2$ and on other potential information computed by the combiner.

Let us now consider a practical example of two competing behaviors. Let's suppose the task at hand is defined as following a sequence of colored lights that determine an unique direction or path[1]. As it has been found, a cardinality of colors of 3 is sufficient to identify a direction of movement (Nouyan et al., 2008). Let the three colors be *white*, *green* and *cyan* (in the given sequence). We can think of having three behaviors (or a single behavior instantiated three times each with a different color) implementing the functionality *GoToLED*. These (or this) behavior will compute the direction and the distance to the closest LED of that specific color and write on the robot state the wheel speeds able to make the robot head towards that LED. Moreover, each behavior can be polled and asked, at any time, what is the direction and the distance to the led computed so far. These three behaviors are examples of behaviors that can be executed in parallel. It is then the duty of the combiner behavior to decide which of the three behavior's output should be propagated in the real robot state. The combiner behavior can implement, for example, the following logic:

1. Set a *current_color* variable to *white*;
2. Execute the three behaviors concurrently;
3. Apply only the output of *GoToLED(current_color)* to the real robot state;
4. Poll the behavior *GoToLED(current_color)* for the distance to the LED;
5. If the distance is below a certain threshold, *current_color=next(current_color)*[2];
6. GOTO 2

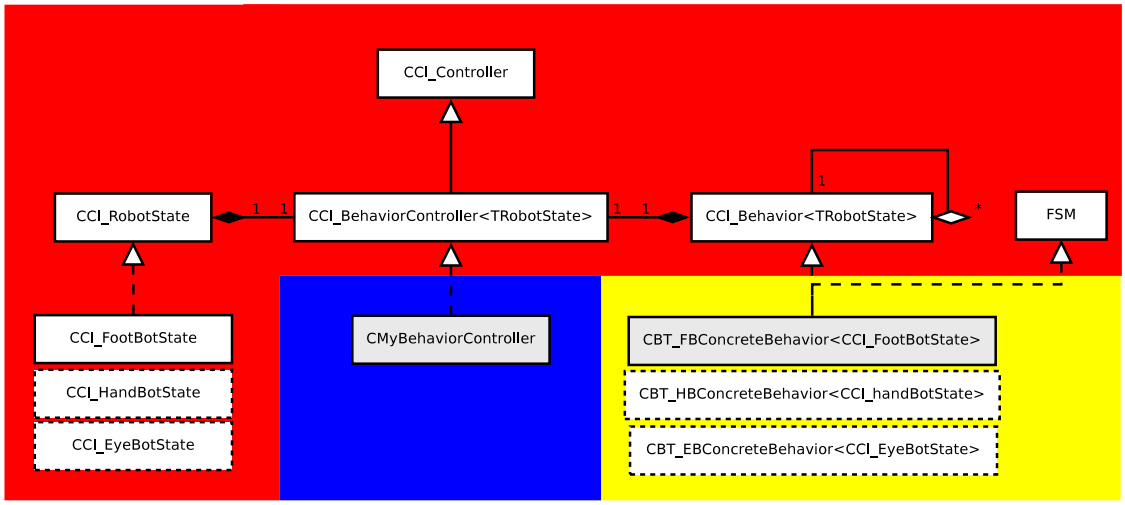


Figure 4.3: The class diagram of the behavioral toolkit inside ARGoS

4.3 Implementation in the Swarmanoid Simulator

Figure 4.3 shows the class diagram of the behavioral toolkit inside the ARGoS simulator. For a comparison with the diagram of the control section of ARGoS without the toolkit refer to Figure 2.6.

As we can see, a behavior controller, instead of extending directly from `CCI_Controller`, extends from a new class called `CCI_BehaviorController`. This new class is a template class. The parameter is the `CCI_RobotState` subtype that refers to the robot for which the controller is going to be implemented. Since three robot types are available in the Swarmanoid project and since sensors and actuators are different in those robots, we provided three subclasses of `CCI_RobotState`: `CCI_FootBotState`, `CCI_HandBotState` and `CCI_EyeBotState`. `CCI_BehaviorController` contains one reference to an object of the `CCI_RobotState` class. It also contains a reference to a `CCI_Behavior` object, which is the root (main) behavior of the controller. A `CCI_Behavior`, in turn, can be composed of multiple other behaviors in a arbitrary way. Furthermore, it also a template class as each behavior will be associated only to a particular robot corresponding to the chosen robot state. Finally, a behavior extends also from the `FSM` class, which is an utility that allows a more modular development of components (states, transitions) inside a single behavior and a better and standardized inter-communication amongst multiple behaviors. It's usage will be better explained in Sections4.4.

The background colors of the class diagram are used to denote the package in which the classes resides. The entire behavioral toolkit resides in the *common* package (red), whereas user written controller go in the *user* package. In the yellow area we have an area denoting

[1]This example is an exemplification of one of the tasks that will be introduced later in Section 5.4.1
[2]where *next(white)=green*, *next(green)=cyan* and *next(cyan)=white*

```
1  int CBTFootBootSampleController::Init(const TConfigurationTree t_tree) {
2    /* Display the configuration tree passed to this controller */
3    CExperimentConfiguration::PrintConfigurationTree(t_tree);
4
5    // Get the a robot state and the root behavior
6    m_pcState = new CCI_FootBotState(GetRobot());
7    m_pcState->Init();
8    m_pcRootBehavior = new CBTFootbotSampleBehavior();
9    m_pcRootBehavior->Init();
10
11   return CCI_Controller::RETURN_OK;
12 }
13
14 void CBTFootBootSampleController::ControlStep() {
15   // Call the control step in of the interface
16   CCI_BehaviorController<CCI_FootBotState>::ControlStep();
17 }
18
19 void CBTFootBootSampleController::Destroy() {
20   // Clean up in the root behavior
21   m_pcRootBehavior->Destroy();
22
23   // Release memory
24   delete m_pcState;
25   delete m_pcRootBehavior;
26 }
```

Figure 4.4: The code of a sample behavior controller

a repository of behavior written by all users in the project. This repository, once available, will reside still in the user package but will be shared amongst all users.

In the following sections we will explain how to write the code of a behavior controller.

4.4 Writing a Behavior Controller

Figure 4.4 shows the C++ code of a behavior controller. As a matter of fact, the code needs not to be written by the user, as its creation is completely automated by a script called `create_behavior_controller.sh`.

In the `Init` method, the main robot state is instantiated (lines 8-9) together with the root behavior (lines 10-11). In the `ControlStep` method, the `ControlStep` method of the superclass is called. Such method's implementation is shown in Figure 4.5. As we can see, this reflect exactly the logic explained in Section 4.2.1 in Figure 4.1: first, the state is *read*, then injected in the top behavior and finally *applied* or written. The `Destroy` method just contains standard cleaning of all instantiated objects.

4.4.1 Writing a Behavior combining Sequential Behaviors

Figure 4.6 shows a simple, atomic behavior (i.e. a behavior that doesn't use any sub-behaviors). We only report the `Step` function, which is the most important one. If this

```
1  template<class TRobotState> class CCI_BehaviorController: public CCI_Controller {
2    . . .
3    /**
4    * @brief The control step of a behavior controller first reads from the sensors
5    * and writes to the state object. Then, it executes the root behavior step, that
6    * might trigger the step of all its sub-behaviors, in order to modify the robot
7    * state. At the end, the modified actuator state is applied using the actuators.
8    *
9    */
10   virtual void ControlStep() {
11     m_pcState->ReadState();
12     m_pcRootBehavior->Step(*m_pcState);
13     m_pcState->ApplyState();
14   }
15   . . .
16 };
```

Figure 4.5: A small subsection of the `CCI_BehaviorController` class

```
1  void CBT_FBObstacleAvoidance::Step(CCI_FootBotState& cRobotState) {
2
3    double fLeftSpeed = 0.0;
4    double fRightSpeed = 0.0;
5
6    double readings[CCI_FootBotState::NUM_PROXIMITY_SENSORS];
7    cRobotState.GetAllProximitySensorReadings(readings);
8
9    // here comes the entire logic which sets
10   // fLeftSpeed and fRightSpeed appropriately
11
12   cRobotState.SetFootBotWheelsAngularVelocity(fLeftSpeed, fRightSpeed);
13 }
```

Figure 4.6: A sample, atomic behavior

```
1  void CBT_FBRandomWalkWithObstacleAvoidance::Init(){
2    m_pcRW = new CBT_FBRandomWalk();
3    m_pcOa = new CBT_FBObstacleAvoidance(CBT_FBObstacleAvoidance::VERSION_VECTOR_BASED);
4  }
5
6  void CBT_FBRandomWalkWithObstacleAvoidance::Step(CCI_FootBotState& cRobotState) {
7    m_pcRW->Step(cRobotState);
8    m_pcOa->Step(cRobotState);
9  }
```

Figure 4.7: A sample behavior combining two other behaviors sequentially: random walk with obstacle avoidance

behavior is indicated as the root behavior of a controller, this step function will be called by the `CCI_BehaviorController Step` method at line 14 of Figure 4.5. Line 7 shows an example of how to poll the sensor readings from a `CCI_RobotState` object, whereas line 12 shows how to write something in the actuators using the same object. Whatever is put in-between these two lines is actual logic of the behavior.

Figure 4.7 shows how a non atomic behavior can be produced through the combination

```
1  void CMyController::ControlStep()
2  {
3    if(!DoubleEq(m_fObstacleAngle,SENSOR_ANGLE[12]) &&
4       !DoubleEq(m_fObstacleAngle,SENSOR_ANGLE[4])){
5         if(m_fObstacleAngle > 5*M_PI/8 && m_fObstacleAngle < M_PI)
6           TurnLeft(MEAN_SPEED);
7         else if(m_fObstacleAngle >= M_PI && m_fObstacleAngle < 11*M_PI/8)
8           TurnRight(MEAN_SPEED);
9         else RandomWalk(MEAN_SPEED);
10   }
11   else RandomWalk(MEAN_SPEED);
12   if(m_fObstacleDistance < 7 && m_bTimeStart == false) {
13    m_nTime = 40;m_bTimeStart=true;
14   }
15   if(m_bTimeStart == true){
16     if(m_nTime !=0) m_nTime--;
17     else {
18       m_bObstaclePresent = false; m_bTimeStart = false;
19     }
20   }
21 }
```

Figure 4.8: A classical controller (i.e. not exploiting the behavioral toolkit) performing random walk with obstacle avoidance

of two other behaviors (atomic or not). This example shares exactly the same logic of the example described earlier in Section 4.2.1. As a matter of facts, combining *Random Walk* with *Obstacle Avoidance* is as simple as this. The current example should be compared with Figure 4.8, which represents a more "classical" way of implementing the same logic in the ARGoS simulator. In the `Init` method (lines 3-4), the two behavior are instantiated, and the *Obstacle Avoidance* constructor is provided with an argument that specifies which version of it should be used, as this behavior is parameterized (as potentially all of them).

As already explained in Section 4.2.1, the simple fact that that two `Step` methods are called in different order ensures a priority relationship between behaviors. However, this method of combining behaviors assumes that such a relationship is known a-priori. In cases where this is not true, priority can still be guaranteed, with the only constraint that behavior written later will always have an higher or equal priority with respect to the one of already existing behaviors that needs to be encapsulated. In this example, the combiner behavior could use the robot state to write to the actuators after lines 10-11, overwriting whatever decision random walk and obstacle avoidance have made. Later, another behavior could include and use `CBT_FBRandomWalkWithObstacleAvoidance` and decide to override whatever decision it makes, and so on ...

4.4.2 Writing a Behavior combining Parallel Behaviors

Figure 4.9 shows an example combiner behavior. The code shows how to implement a behavior that enables a robot to follow a light chain. The logic behind this has already been

```
1  int CBTFootbotFollowLEDChainParallel::Init() {
2    ...
3    m_unCurrentChain[0] = COLOR_WHITE;
4    m_unCurrentChain[1] = COLOR_GREEN;
5    m_unCurrentChain[2] = COLOR_CYAN;
6
7    for (unsigned int i = 0; i < 3; i++) {
8      m_pcFollowLEDs[i] = new CBTFootbotFollowCeilingLED(m_unCurrentChain[i]);
9    }
10 }
11
12 void CBTFootbotFollowLEDChainParallel::Step(CCI_FootBotState& cRobotState) {
13   // Duplicate the state three times
14   m_pvParallelStates[0] = m_pvParallelStates[1] = m_pvParallelStates[2] = cRobotState;
15
16   // Step parallely in all the behaviors
17   for (unsigned int i = 0; i < 3; i++) {
18     m_pcFollowLEDs[i]->Step(m_pvParallelStates[i]);
19   }
20
21   // Copy back in the main state the state modified by the appropriate behavior
22   cRobotState = m_pvParallelStates[m_nNextColorIndex];
23
24   // Check whether to change the behavior that must be considered
25   double minBlobDistance = m_pcFollowLEDs[m_nNextColorIndex]->GetDistanceToLEDColor();
26   if (minBlobDistance < DISTANCE_THRESHOLD) {
27     m_nNextColorIndex = (m_nNextColorIndex + 1) % 3;
28   }
29 }
```

Figure 4.9: A sample behavior combining two other behaviors parallely

explained in Section 4.2.2.

Assume we have implemented a behavior called `CBTFootbotFollowCeilingLED` that is instantiated three times by providing each time a different color parameter (line 10). The combination logic is implemented in the `Step` method: we first replicate the main robot state three times (line 18); we then execute the three behaviors in parallel (line 22); we then then "combine" then by taking into account only the behavior's output corresponding to the color that is currently being followed (line 26); finally, if the robot is close enough to the LED, we go to the next color (lines 29-32).

The code in Figure 4.9 is an example to show how to combine parallel behaviors. A controller using such behavior does not pretend to be the most efficient solution to the given task. In this particular case, we do not need to step in all three sub-behaviors since only the output and the internal state of the behavior corresponding to the current color is taken into account. However, a logic similar to the one shown in Figure 4.9 can be used in the more general case where multiple behaviors' output or internal state need to be checked before making a decision.

In the example we note that combiner behaviors need often to access some information about the "status" or "internal state" of the behaviors they are combining. In the example (line 29), we used a method that has the role of providing the specific information needed in this

```
CBTFootbotSampleBehavior::Init() {
  // Initializing the states
  m_pcStateOne = initializeState(this, STATE_ONE, "[STATE_ONE]");
  m_pcStateTwo = initializeState(this, STATE_TWO, "[STATE_TWO]");
  m_pcStateThree = initializeState(this, STATE_THREE, "[STATE_THREE]");

  // Define all the transitions conditions
  m_pcStateOne->addTransition("conditionX", m_pcStateTwo);
  m_pcStateTwo->addTransition("conditionY1", m_pcStateThree);
  m_pcStateThree->addTransition("conditionY2", m_pcStateTwo);

  // Set initial state
  setStartState(m_pcStateOne);
}

void CBTFootbotSampleBehavior::Step(CCI_FootBotState& cRobotState) {
  . . .
  if(GetStateID() == STATE_ONE){
    // Deal with STATE_ONE
    input(TransitionOneToTwo(cRobotState));
  }
  else if(GetStateID() == STATE_TWO){
    // Deal with STATE_TWO
    input(TransitionTwo2Three(cRobotState));
  }
  else if(GetStateID() == STATE_THREE){
    // Deal with STATE_THREE
    input(TransitionTwo2Three(cRobotState));
  }
}

string CBTFootbotSampleBehavior::TransitionOneToTwo(CCI_FootBotState& cRobotStateCopy){
        string result = "";
        // if (conditions met to change to STATE_TWO) result = "conditionX";
        return result;
}

string CBTFootbotSampleBehavior::TransitionTwo2Three(CCI_FootBotState& cRobotStateCopy){
        string result = "";
        // if (conditions met to change to STATE_THREE) result = "conditionY1";
        // else if (conditions met to change to STATE_TWO ) result = "conditionY2";
        return result;
}
```

Figure 4.10: A sample template behavior showing how to use the FSM tool

case, i.e. the distance to the closest LED. In the next section, we introduce the Finite State Machine (FSM) tool, which enables both a more modular coding of an individual behavior and a more standardized inter-behavior communication mechanism.

4.4.3 Dealing with Finite State Machines

Behaviors can be thought as software modules that provide some sort of mapping between stimuli coming from the real world and robot's actions. Such mapping can be formalized using a Finite State Machine (henceforth FSM) (Minsky, 1967).

FSMs are not only a useful mathematical tool to model such systems, but they can also be used to provide a better logical and structural organization of real code. In the behavioral

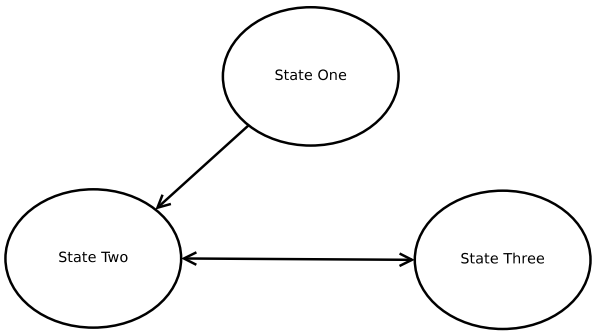


Figure 4.11: The diagram of the very simple FSM implemented in Figure 4.10

toolkit, we included such a tool obtained by adapting the FSMPP library, which is freely available library[3] to build Deterministic Finite State Machines in C++.

Suppose we have a behavior whose corresponding FSM is the one depicted in Figure 4.11. The corresponding code in the behavioral toolkit is shown in Figure 4.10.

The code has an organized structure that allows the separation amongst code pertaining to different states and between states and transitions. In the example, the three states are initialized in the `Init` function (lines 3-5), where each state is associated to both an integer unique identifier and to a string, which is useful for debugging purposes. In the following three lines (lines 8-10), we link states through transitions. Strings such as "conditionX" are labels that identify events that triggers transitions. The last line of the `Init` function is used to specify the initial state.

In the `Step` method, a simple `if` or alternatively `switch` construct can be used to deal with the different states. After dealing with each state, the `input` method needs to be called: it accepts a string argument, which represents the label of a possible event that can cause a transition. Code that manages transitions can be organised in specialized methods. In the example, method `TransitionOneToTwo` (lines 32-36) menages unidirectional the transition between *State One* and *State Two*, whereas method `TransitionTwo2Three` (lines 38-43) menages bidirectional the transition between *State Two* and *State Three* (notice the conventional use of "to" or "2" to distinguish between the unidirectional and bidirectional case).

The final example in Figure 4.4.3 shows how to use the FSM tool in order to achieve a more standardized communication between different behaviors. In the example, a combiner behavior $B3$ is trying to combine two behaviors, $B1$ and $B1$, which both implements the FSM interface. As a result, the `GetStateID` method can be used to query sub-behaviors about their internal state (lines 4 and 7).

In this section we provided simple examples only. In the next chapter, we will describe more complex behaviors. All behaviors were are implemented using the behavioral toolkit,

[3]http://sourceforge.net/projects/fsmpp, under by the GPS licence

```
1  void B3::Step(CCI_FootBotState& cRobotState)
2  {
3    . . .
4    if(m_pcB1->GetStateID() == B1::STATE_X){
5      // some logic here
6    }
7    if(m_pcB2->GetStateID() == B2::STATE_X){
8      // some other logic here
9    }
10 }
```

Figure 4.12: A sample template behavior showing how to use the FSM tool to query behaviors about their state

and almost all of them are organized using the FSM tool.

Chapter 5

Experiments with Modular Behaviors

In order to test its effectiveness, the behavioral toolkit has been used to develop several behaviors for the *foot-bot* and for the *hand-bot*. In this chpater, we present several experiments employing these behaviors.

The chapter is organized as follows: in Section 5.1.1 we define the overall task and we show the overall organization of the solving global behavior; subsequently, in Section 5.3 and Section 5.4, we describe the two main sub-behaviors composing the global one.

5.1 Experiment Definition

The task we are tackling is the assembly and control of a composite robot called *phat-bot*. The phat-bot (shown in Figure 5.1) is a robot which is composed by three foot-bots assembled to an hand-bot. The aim of such a robot is to provide a platform able to navigate and perform manipulation at the same time, which is not possible by using foot-bots and hand-bots alone.

5.1.1 High Level Description of the Behaviors

Clearly, the task we are facing can be subdivided in two: assembly of a phat-bot and collective navigation. In the following, we first introduce simple behaviors implemented on foot-bots only. The combination of such behavior will produce the assembly behavior. Subsequently, we describe behaviors able to achieve interaction through a simple communication mechanism amongst multiple robots, both homogeneous (footbots-footbots) and heterogeneous (footbots-handbots). The combination of these behaviors with the assembly behavior produces a top behavior able to perform assembly and navigation with obstacle avoidance in an environment with obstacles.

The top behavior dependency structure is depicted in Figure 5.2, where a directed edge denotes the relationship “uses” or “is composed by”. As we can see, they are organized in

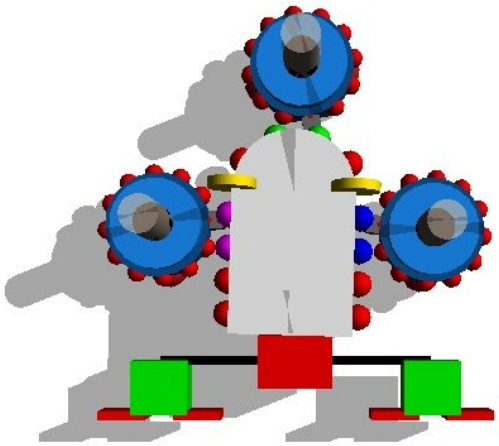


Figure 5.1: A simulated phat-bot in all its beauty

three level of competence. At the lowest level, we can find basic behaviors that consist in the building blocks that are reused multiple times by several other behaviors: *random walk*, *obstacle avoidance*, *go to LED* and *go to light*. These will be explained in Section 5.2. At the intermediate level, we find two behaviors that are used to assembly and to navigate with obstacle avoidance: *assemble* and *push/pull*. These behaviors will be described in more detail in the following two sections 5.3 and 5.4. Finally, the highest level behavior, *assemble and move*, simply interconnects assembly and navigation. On the diagram, we also show the interdependency between all the behaviors except *random walk* and an utility class: such class, better described in Section 5.2.5, provides functionnalities that are common to multiple behaviors but that cannot be considered behaviors in themselves.

5.1.2 Experimental Setup

All the behaviors and experiments were implemented in ARGoS, the Swarmanoid simulator introduced in Section 2.3.

More into the details, the simulation was executed using the OpenGL visualization or no visualization at all, and with the *2D Simplified Kinematics* Physics engine (Figure 2.4). Inside this physics engine, we implemented and adapted the physics model and the collision model of the hand-bot, the foot-bot and of the compound entity that is created once foot-bots assemble with an hand-bot.

However, the use of such the above-mentioned physics engine puts a number of simplifying assumptions on the task at hand. Such assumptions can make the current simulation potentially not sufficient for having a controller which is directly transferable in to the real robots.

The most relevant simplification due to the current physical simulation is explained in the

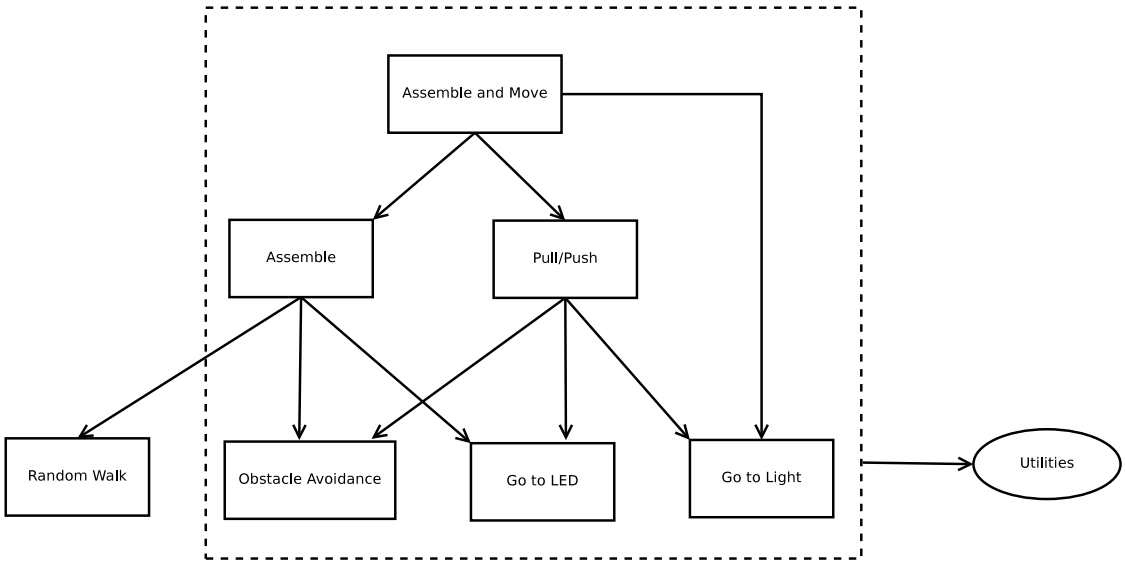


Figure 5.2: The overall organization of the *assemble and move* behavior

following. Once the phat-bot is assembled, each foot-bot rotates the wheels as they were not linked in a compound and a vector corresponding to their individual motion is computed from the wheels speeds. Given all those vectors in the compound, an average vector is computed and directly applied to obtain the compound motion of the phat-bot. As a result we obtain that, during the compound motion, the relative orientation of the wheels of each individual foot-bot and the corresponding friction is not taken into account at all. In reality, if the wheels are not oriented in the same direction of the phat-bot motion, their friction could potentially prevent the entire entity from moving or at least slow the motion down. To account for this, each foot-bot includes a turret that can be actuated in order to change the relative rotation of the wheels. A more complex controller which also includes such turret's actuation is left out for future work.

5.2 Basic Behaviors

Before explaining the logic behind the *assemble* behavior, we briefly introduce the basic behaviors that were composed in order to produce it.

5.2.1 Random Walk

The random walk behavior is useful to make a mobile robot explore the environment in a uniform way. It consists in applying some stochasticity to the wheels motion. Our implementation is based on the following, very simple idea:

1. We generate a tuple v, t_r, t_t, d, where v is a random velocity scalar, t_r is a random

rotation time, t_t is a random translation time and d is a random, binary rotation direction. Each of these are uniformly drawn from a different, bounded interval;

2. we rotate the robot clockwise or anti-clockwise (according to d) for t_r timesteps;
3. we move the robot forward for t_t timesteps at v speed;
4. go to 1;

5.2.2 Obstacle Avoidance

Our obstacle avoidance behavior can be parameterized at instantiation time, as already shown in Figure 4.7. The parameter is an integer number that uniquely identify which version of obstacle avoidance the user is willing to use. So far, we implemented two versions:

Dummy We use the proximity sensors to check whether there is a reading above a certain threshold. If this is the case, we simply take the angle corresponding to the highest sensor reading and we turn the robot clockwise or anti-clockwise according to this angle.

Vector-Based For each of the proximity sensor readings, we compute a vector according to the angle reading. We then sum up these vectors, filter the resulting vector, compute the opposite vector and convert it into an appropriate wheel motion. This makes use of two utility functions, i.e. `FilterVector` and `SetWheelsSpeedFromVector`, which will be better explained in Section 5.2.5.

5.2.3 Go to LED

The *go to LED* is also a parametric behavior, where the parameter represent one of the possible LED colors. This behavior is simple and it works as follows. It uses the robot's omnidirectional camera to collect all blobs of the same color as the one determined by the parameter. It then computes the distance and the direction of the closest amongst these blobs. Finally, it converts these two numbers into a vector, it filters it since there can be noise and it applies it to the wheels. Also this behavior uses the utility functions `FilterVector` and `SetWheelsSpeedFromVector`, explained in Section 5.2.5.

5.2.4 Go to Light

This behavior is very similar to *go to LED* and even simpler. There are only two differences. The first difference is the absence of a color parameter. The second difference is due to the sensor itself: differently from the camera, the light sensor can only return a gradient of intensity of light, that is equivalent to saying that it is not possible to distinguish clearly

between two different light sources and hence, by knowing the angle corresponding to the highest intensity, we also automatically know only the direction to the closest and/or brightest light source. This information is then translated into the corresponding wheels motion.

5.2.5 Utility Class

The utility class does not represent a behavior in itself but just a collection of utility functions. So far, the utility functions we provided (which are mainly intended for the foot-bot) are the following:

FilterVector This function can be used to filter a 2D vector using a mobile average filter. The filtered vector is computed as follows:

$$\bar{\mathbf{V}}_t = \alpha \mathbf{V}_t + \frac{(1-\alpha)}{W}\mathbf{V}_{t-1} + \frac{(1-\alpha)}{W}\mathbf{V}_{t-2} + \ldots + \frac{(1-\alpha)}{W}\mathbf{V}_{t-W} \tag{5.1}$$

where $\mathbf{V}_t$ is the vector at time t, $\mathbf{V}_{t-k}$ is the vector delayed k steps in the past, $W \in \mathbb{N}$ is the filtering window or memory size and $\alpha \in (0, 1)$ is a recency factor. W and α can be changed at any time dynamically. Lower values of α (more influence of the past) and higher values of W (more memory) makes the filter stronger, and vice-versa.

SetWheelsSpeedFromVector This function is used to convert a vector into two floating point number that represent the speed to apply to the wheels. This function comes in three versions: *no reverse*, *with reverse* and *reverse only*. The first version computes the difference between the robot's heading direction and the vector direction and applies, if necessary, a turning speed to the wheels (whose intensity depends on the difference) in order to minimize this difference. In the *with reverse* version, if the difference between the two angles is too high, then the robot is allowed to move backwards also. Finally, the *reverse only* version works as the *no reverse* but the minimized difference is the one between the vector's angle and the opposite of the robot's heading (the robot's back), yielding to a robot that will always move backwards.

GetProximityAngle This function assumes we know the maximum amount of proximity sensors installed on the robot. Given this number and and an index provided as input, the function can be used to compute the angle of the proximity sensor corresponding to that index.

5.3 Phat-bot Assembly

The assemble behavior was obtained by combining the three basic behaviors *random walk*, *obstacle avoidance* and *go to LED* by the use of a very simple finite state machine, which

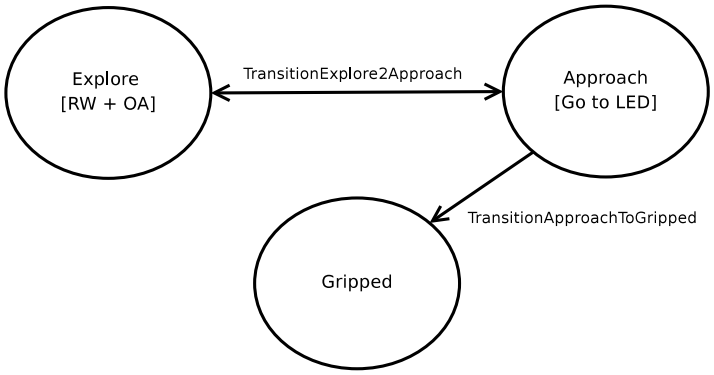


Figure 5.3: The FSM of the of the *assemble* behavior

is depicted in Figure 5.3. Each state is represented with a name and with the name of the behaviors it is using between square brackets. As we can see, it is composed by three states only:

Explore It combines *random walk* with *obstacle avoidance* in a sequential fashion as already explained in Figure 4.6 of Section 4.4.1. Foot-bots in this state will explore the environment in order to search for a hand-bot.

Approach It uses the *go to LED* behavior parametrized with some color in order to approach a colored slot lit on the hand-bot's LED ring while aligning to it. The colored slot can be of three different colors, as explained in the following.

Gripped This state is mainly used for communicating to an potential top behavior the fact that the foot-bot has finished assembling (refer to Figure 4.4.3 in Section 4.4.3).

The critical part of the *assemble* behavior is the `TransitionExplore2Approach` method, which implements the logic of the bidirectional transition between the two mentioned states. Such transition must be clever enough to allow a foot-bot to start approaching an hand-bot only when some conditions are satisfied and stop approaching it in case these conditions are no longer satisfied. Such desired conditions are:

- The foot-bot can approach only from some predefined angles, i.e. from the back, from the right and from the left and not diagonally;
- The foot-bot can approach only if there is no other foot-bot approaching from the same angle.

Futhermore, we want also each assembled foot-bot to know its role (i.e. where it assembled) in the compound structure in order to facilitate navigation later. In order to guarantee the above properties, we defined a very simple LED color protocol. It works as shown in Figure 5.4. The hand-bot will turn on its LED as shown in Figure 5.4a: blue, green and violet

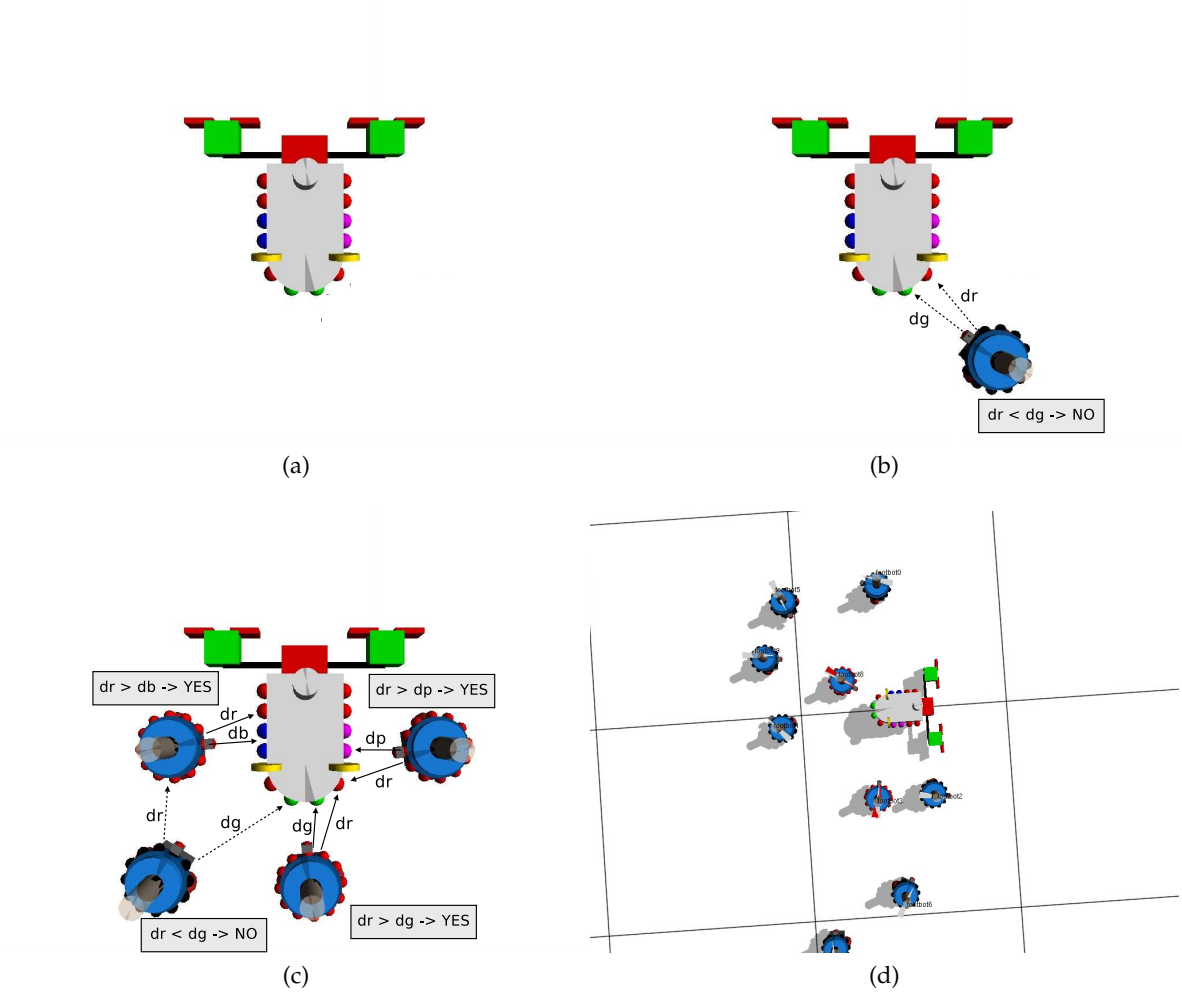


Figure 5.4: The assembly behavior: (a) The assembly protocol on the hand-bot, (b) foot-bot cannot assemble since the angle is not correct, (c) foot-bot cannot assemble since another foot-bot is already assembled at that slot and (d) a crowded situation where foot-bots still manage to assemble. Notation: d_r = distance to closest red blob, d_g = distance to closest green blob, etc ...

colors defines connection areas or slots, whereas the red color defines areas where foot-bots are not allowed to connect. The three colors denotes the left, back and right slot, which will be also used to determine the foot-bot role as we will see later in Section 5.4. Also, the area which is the closest to the gripper is lit with two red LEDs instead than one: this is to prevent a foot-bot attaching too close to the hand-bot arms, which would saturate the proximity sensors and make the navigation task more difficult. In the `TransitionExplore2Approach` method we use the following rule: if the distance to the closest red color is greater than the distance to the closest slot we trigger the transition from explore to approach, otherwise we trigger the transition from approach to explore. This simple rule can guarantee both of the

above properties. In fact, as we can see from Figure 5.4b, a foot-bot that is misaligned cannot enter the approach state since it will see the red closer than any of the other slots. On the other hand, a foot-bot entering the approach state will also turn all its LEDs to red. In this way, we disallow a foot-bot to enter the approach state in case another foot-bot is already approaching, since also in this case red will be seen as closer than the slot (Figure 5.4c).

The above behavior has been tested and it ensures that only three robots attaches to the hand-bot also in very critical and crowded situations like the one shown in Figure 5.4d.

The behavior can be improved in several ways. Most notably, we need to investigate how to use less colors (ideally only two instead of four) in order to define the slot areas. Additionally, right now we only rely on *random walk* for search for an hand-bot and for a suitable slot. Behaviors such as *go around* could be useful to search fo a slot once an hand-bot has been found, in order to increase efficiency.

5.4 Phat-bot Navigation

In this section we describe the cooperative navigation strategies that we implemented on the foot-bot and on the hand-bot in order to let it move in an environment without and with obstacles. In Section 5.4.1, we first briefly introduce the *follow chain* behavior, which is a variation of the behavior already explained in Sections 4.2.2 and 4.4.2 able to let a phat-bot follow a chain made of colored LEDs. Subsequently, in Section 5.4.2, we describe the *push* and *pull* behaviors, which are used to let a phat-bot move towards a direction while avoiding obstacles.

5.4.1 Follow Chain Behavior

A sample *follow chain* behavior was already explained in Section 4.4.2. However, for efficiency purposes, in our actual experiments we implemented a slightly different behavior. These two behaviors differ mainly in the following two key points:

- The “efficient” version does not use three *go to led* behaviors but just considers the direction and the distance to the led of the color that is currently being followed;
- The version presented earlier works on an individual foot-bot without assuming its connection to a phat-bot. Instead, the modified version assumes the connection with the phat-bot, hence it is executed only if the foot-bot is connected to the back slot.

5.4.2 The Push and Pull Behaviors

The push and pull behaviors are very similar and share the basic idea and most of the implementation details. The main difference between the two behaviors is that push requires

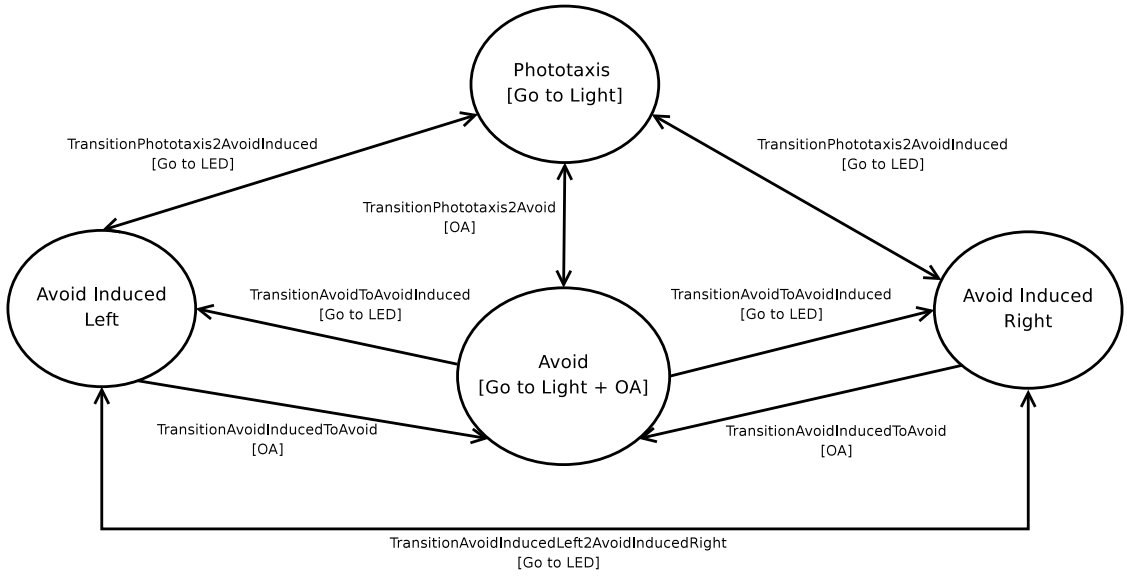


Figure 5.5: The FSM of the of the *pull* and *push* behaviors

the cooperation of the hand-bot whereas pull does not. The requirements on both behaviors are the following: given a common direction of movement given by the presence of a light source, the foot-bot must be able to move the hand-bot towards that direction while avoiding obstacles, either by pushing or pulling.

The task is somehow challenging since the three foot-bots need to coordinate their movements both to move towards the common direction and to avoid obstacle. The first part of the task has been simplified by allowing the light source to be perceived by all robots, together with the assumptions related to the physics engine and explained already in Section 5.1.2. This leaves only the second part of the problem completely unsolved: how to coordinate the robots movement while doing obstacle avoidance?

We solved the above issue by using, again, a colored LEDs protocol. The basic idea is the following: if a foot-bot sees an obstacle, it should use its LEDs to suggest to the other foot-bots to translate the phat-bot either left or right (we used cyan to denote right and white to denote left). If a foot-bot does not perceive an obstacle but perceives either a cyan or a white color blob, it should translate the phat-bot either left or right as instructed. Additionally, if the foot-bot perceiving the cyan or white blob is the central one, it should also act as a relay by turning its LEDs to the same color too (since left and right attached foot-bot cannot perceive themselves due to the hand-bot in the middle).

The entire idea is again summarized by a very simple finite state machines, as depicted in Figure 5.5. Again, each state is characterized by a label denoting its name and a list of behaviors it is using between square brackets. This time, also transitions are using behaviors, hence they are summarized as well in square brackets after the name of the transition.

The meaning of each state and the corresponding logic can be summarized as follows:

Phototaxis No obstacles are perceived. Hence, each robot can perform the phototaxis by stepping into the *go to light* behavior, causing the entire phat-bot to move towards the direction of the light. The difference between push and pull is that a different version of the `SetWheelsSpeedFromVector` utility function is used, i.e. no reverse and with reverse respectively (refer to Section 5.2.5 for more details).

Avoid The obstacle has been directly perceived. Hence, if the foot-bot is attached to one of the two sides of the hand-bot, it starts "pushing" in order to translate the phat-bot away from the obstacle, else (it is attached to the center) it doesn't apply any motion to the wheels. Furthermore, if the foot-bot is attached to the left, it should turn LEDs into white to tell the other foot-bots to translate the entity to the right. Vice-versa, if attached to the right, it should turn cyan. Finally, if the foot-bot is attached to the center, the correct color must be determined by checking the proximity sensor's angle at which the obstacle has been perceived together with the angle of the light (this is the point where the two sub-behaviors *go to light* and *obstacle avoidance* are used in this state).

Avoid Induced Left or Right The robot did not detect an obstacle but it did perceive a signal by another foot-bot which detected an obstacle. At this point, the correct course of action is detected by the $\langle s, r \rangle \in \{left, right\}^2$ pair, where s is the state (induced left or right) and r is the role (attached to the left or to the right). If $s = r$, the correct action is to move backwards, whereas it should move forward in case $s \neq r$. As an example, if $s = left$ (another foot-bot has seen an obstacle on the right of the phat-bot and hence it should translate to left) and $r = right$, it should move forward. The central foot-bot has, as we already said, only the role of propagating the signal, hence it should turn cyan or white according to whether it's in the avoid induced left or right state respectively.

Furthermore, the transitions can be explained as follows:

TransitionPhototaxis2Avoid This transition uses the *obstacle avoidance* behavior to check whether an obstacle is present or not. If the obstacle is present, the transition from the phototaxis state to the avoid state is triggered, otherwise the opposite transition is triggered. The *obstacle avoidance* behavior can be parameterized in order to specify an activation area, i.e. which set of reading to consider and which to discard. This is done since each foot-bot needs to discard reading coming from the circular section which faces the hand-bot, as shown in Figure 5.6.

TransitionPhototaxis2AvoidInduced This transition uses the *go to led* behavior in order to check whether there is a cyan or white color blob in range. If this is the case, the

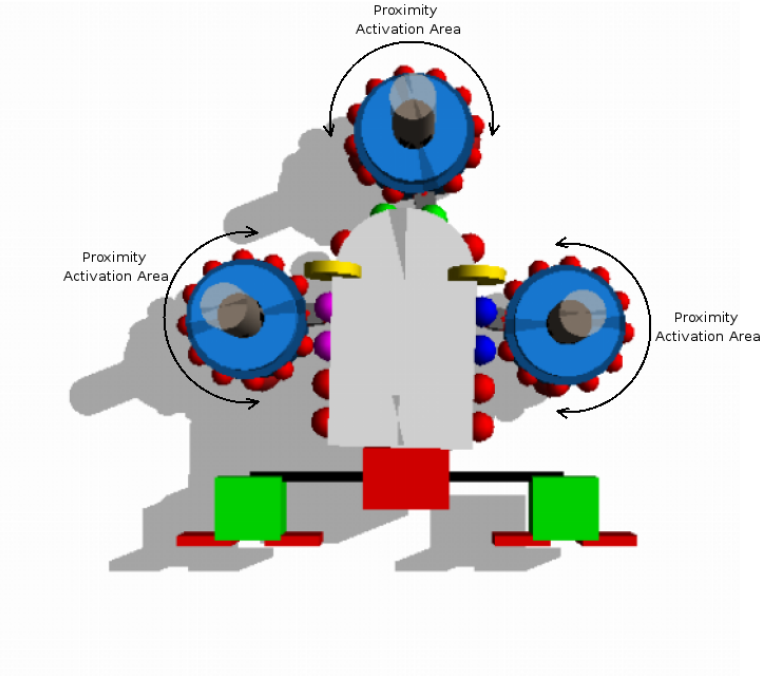


Figure 5.6: Diagram showing the range of activation of the proximity sensors in the *obstacle avoidance* behavior during phat-bot navigation

transition from phototaxis to avoid induced (left or right respectively) is triggered (some other foot-bot has perceived an obstacle), otherwise the opposite transition is triggered (no more obstacles are perceived, hence the foot-bot can go back to phototaxis state).

TransitionAvoidToAvoidInduced This transition is defined only in the pull behavior and in case the foot-bot is attached to the back slot. It uses the *go to led* behavior to understand whether another foot-bot has perceived an obstacle on the left or on the right. If this is the case, switching to the avoid induced state has an higher priority since if we do not translate the phat-bot in the correct direction it will get stuck for sure.

TransitionAvoidInducedToAvoid This transition uses the *obstacle avoidance* behavior to detect obstacles as in the `TransitionPhototaxis2Avoid` transition. This transition is needed since the foot-bot that perceives an obstacle directly needs to react in order to avoid getting stuck.

TransitionAvoidInducedLeft2AvoidInducedRight If a robot is in the avoid induced left state and perceives a white signal through the *go to led* behavior, it should react and change to the avoid induced right state (vice-versa for the other case).

We can notice that the difference between the push and the pull behavior is minor. The main difference is due to the fact that, in the push behavior, the hand-bot needs to cooperate as

well. As said in Section 2.2.3, the hand-bot is provided with proximity sensors in each gripper. These sensors can be used to detect obstacles as well. To achieve so, we implemented a very simple behavior on the hand-bot: if the hand-bot perceives something close to its left gripper, it turns white, whereas if it perceives an obstacle close to its right gripper it turns cyan. The behavior of the foot-bot can be kept the same, since it automatically integrates the presence of the hand-bot due to the standardized color protocol.

In order to compare the two strategies, we ran a simple experiment that consisted in a race between phat-bots using the two strategies. Each strategy was executed for 100 runs in an environment like the one depicted in Figure 5.7a. A run is completed when the phat-bot reaches the end of the corridor (not shown on the figure). Figure 5.8 shows there is no substantial difference between the performances (number of time-steps to complete the task) of the two behaviors, and that both exhibit few outliers showing that the phat-bot might get stuck some times while trying to overcome some obstacles.

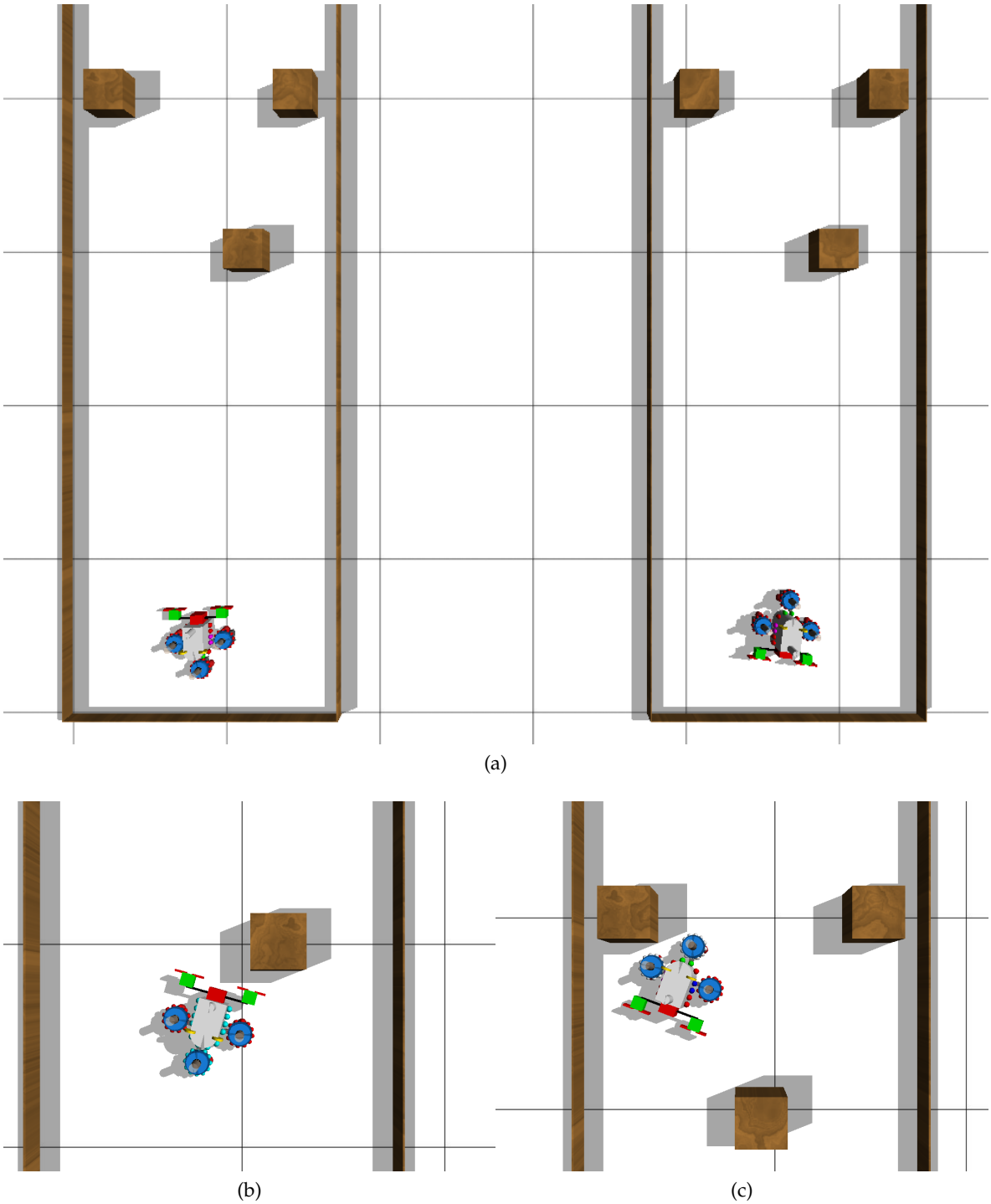


Figure 5.7: The *push* versus the *pull* behavior: (a) the two behaviors competing in the same environment, (b) the push avoiding an obstacle on the right and (c) pull avoiding an obstacle on the left

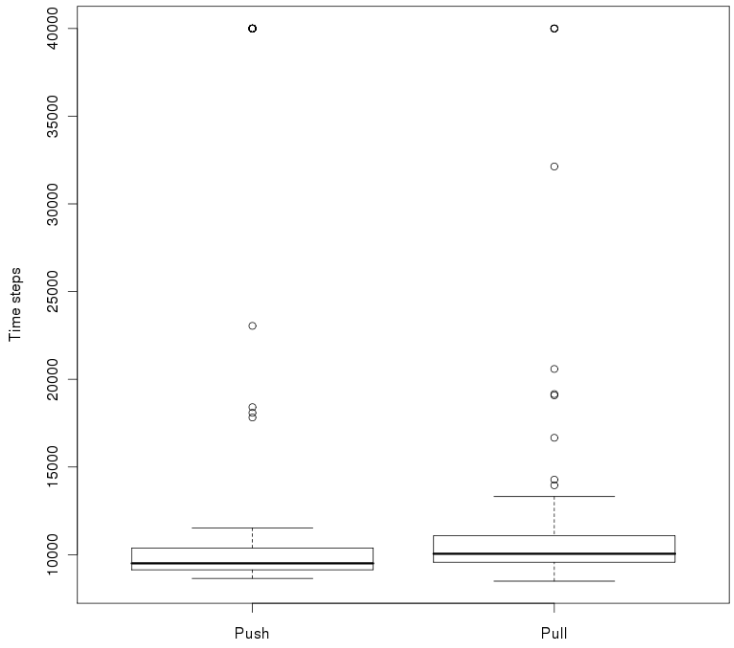


Figure 5.8: Box and wisker plot comparing the time delays of *push* and *pull* behaviors, showing there is no substantial difference between the two. Some outliers are present in both cases.

Chapter 6

Conclusions and Future Work

In this work, we presented an architecture that allows the development of behaviors while respecting software engineering principles like code re-usability, modularity, separation of concerns, abstraction and incremental development. We showed that the proposed architecture is inherently modular: thus, it allows to combine behaviors together and to reuse them when needed. The architecture exploits the capabilities of object oriented languages, an makes it possible to write pieces of code that are more readable thanks to a standardized mechanism of interconnection and communication between both multiple behavior and inside a single behavior. The architecture has been validated through an experiment which involved (a) assembly of a composite robot made of two different types of robot and (b) collective navigation and obstacle avoidance using this new composite robot.

We believe that this work can be extended and improved in several ways. Amongst those, we identified two main categories of improvements, which could yield us closer to the vision dream of having a complete architecture for swarm robotics.

Scripting Behaviors In Christensen et al. (2007), they proposed a scripting language called SWARM-MORPH for the development of behaviors for robots composing the swarm-bots project (Mondada et al., 2004). Although very promising, such scripting language is also limited since it does not allow to develop modular primitives (basic building blocks of the scripting language). Hence, a first interesting direction of future works would be to integrate an extended SWARM-MORPH script into the behavioral toolkit. By having a one to one mapping between behaviors and primitives, it would be possible to have scripting building blocks that are no longer primitives but composite elements composed of different primitive or non-primitive elements.

Adaptive Behaviors A further step to go towards an architecture for swarm robotics is to extend the current architecture in a way to include adaptability. There are many possible ways to achieve this, which include:

- Including the response threshold model (Granovetter, 1978; Bonabeau et al., 1997) into the FSM (Section 4.4.3) together with a mechanism to acquire the social stimulus from the local (both social and non-social) environment;
- Develop a mechanism for the auto-composition of behaviors inside an individual robots. This could be done, for example, by using evolutionary techniques (Togelius, 2004; Ashlock, 2006).
- Use more advanced optimization and learning techniques to guarantee more adaptive behaviors. Examples would be particle swarm optimization (Kennedy and Eberhart., 2001) or policy search reinforcement learning techniques (Peshkin, 2002).

Going towards an architecture for swarm robotics that allows the development of swarm-level controllers while observing software engineering principles is a very challenging goal. The two main difficulties are represented by: the high heterogenuity and dynamicity of both hardware and software solutions available to robotics; the absence of a predetermined recipes to translate individual-level behaviors to global-level behaviors. We believe that, in the present work, we developed and applied some general principles that can be used to tackle, at least partially, the first of these two issues. Extensions of the proposed work that can also tackle the second issue are very interesting topics for future works.

Bibliography

Abraham, A., Grosan, C., and Ramos, V., editors (2006). *Swarm Intelligence in Data Mining*. Springer, Berlin / Heidelberg.

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337.

Arkin, R. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112.

Arkin, R. (1998). *Behavior-Based Robotics*. MIT Press, Cambridge.

Ashlock, D. (2006). *Evolving Finite State Automata*, chapter 6, pages 143–166. Evolutionary Computation for Modeling and Optimization. Springer, New York.

Ben-Jacob, E., Cohen, I., and Levine, H. (2000). Cooperative self-organization of microorganisms. *Advance in physics*, 49(4):395–554.

Beni, G. (2004). From swarm intelligence to swarm robotics. In *Proceedings of the Workshop on Swarm Robotics, 8th International Conference on Simulation of Adaptive Behavior*, Los Angeles, CA. Springer.

Beni, G. and Wang, J. (1989). Swarm intelligence in cellular robotic systems. In *Proceedings of the NATO Advanced Workshop on Robots and Biological Systems*, Tuscany, Italy. Springer.

Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York.

Bonabeau, E., Sobkowski, A., Theraulaz, G., and Deneubourg, J.-L. (1997). Adaptive task allocation inspired by a model of division of labor in social insects. In *Biocomputing and emergent computation: Proc. of BCEC97*, pages 36–45. World Scientific Press.

Bonasso, R. (1991). Integrating reaction plans and layered competences through synchronous control. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

Brooks, R. (1990). Elephants don't play chess. *IEEE Journal of Robotics and Automation*, 6:3–15.

Caro, G. D. and Dorigo, M. (1998). Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365.

Christensen, A., O'Grady, R., Birattari, M., and Dorigo, M. (2008). Fault detection in autonomous robots based on fault injection and learning. *Autonomous Robots*, 24(1):49–67.

Christensen, A., O'Grady, R., and Dorigo, M. (2007). Morphology control in a multirobot system. *IEEE Robotics Automation Magazine*, 11(6):732–742.

Crespi, V., Galstyan, A., and Lerman, K. (2008). Comparative analysis of top-down and bottom-up methodologies for multi-agent systems. *Autonomous Robots*, 24(3):303–313.

Detrain, C. and Deneubourg, J.-L. (2006). Self-organized structures in a superorganism: do ants "behave" like molecules? *Physics of Life Reviews*, 3(3):162–187.

Dorigo, M. and Şahin, E. (2004). Guest editorial. special issue: Swarm robotics. *Autonomous Robots*, 17:111–113.

Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. MIT Press, Cambridge, MA.

Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T. H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D., and Gambardella, L. M. (2004). Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2–3):223–245.

Espiau, B., Kapellos, K., and Jourdan, M. (1995). Formal verification in robotics: Why and how? In *in The International Foundation for Robotics Research, editor, The Seventh International Symposium of Robotics Research*, pages 20–1. Press.

Estlin, T., Gaines, D., Chouinard, C., fisher, F., no, R. C., Judd, M., Anderson, R., and Nesnas, I. (2005). Enabling autonomous rover science through dynamic planning and scheduling. In *Proc. of the IEEE Aerospace Conference (Big Sky)*, pages 385–396, Piscataway. IEEE Press.

Frigg, R. and Hartmann, S. (2006). Models in science. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*.

G. Butler, A. Gantchev, P. G. (2001). Object-oriented design of the subsumption architecture. *Software: Practice and Experience*, 31(9):911–923.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, volume 2, pages 70–74, New York, NY, USA. ACM.

Gautrais, J., Michelena, P., Sibbald, A., Bon, R., and Deneubourg, J.-L. (2007). Allelomimetic synchronisation in merino sheep. *Animal Behaviour*, 74:1443–1454.

Gerkey, B. and Mataric, M. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9):939–954.

Granovetter, M. (1978). Threshold models of collective behavior. *American Journal of Sociology*, 83(6):1420–1443.

Groß R. and Dorigo, M. (2007). Fifty years of self-assembly experimentation. In Shen, W.-M., Lipson, H., Stoy, K., , and Yim, M., editors, *Proc. of the Workshop on Self-Reconfigurable Robots/Systems and Applications*, Marina del Rey, CA. USC Information Science Institute.

Groß R. and Dorigo, M. (2008). Evolution of solitary and group transport behaviors for autonomous robots capable of self-assembling. *Adaptive Behavior*, 16(5):285–305.

Grünbaum, D., Viscido, S., and Parrish, J. K. (2004). Extracting interactive control algorithms from group dynamics of schooling fish. *Lecture Notes in Control and Information Sciences*, 309:103–117.

Jacobi, N. (1997). Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics. In Husbands, P. and Harvey, I., editors, *Proceedings of the Fourth European Conference on Artificial Life: ECAL97*, pages 348–357. MIT Press, Cambridge, MA, USA.

Kennedy, J. and Eberhart., R. (2001). *Swarm Intelligence*. Morgan Kaufmann.

Kortenkamp, D., Bonasso, R., and Murphy, R. (1998). *Artificial Intelligence and Mobile Robots*. AAAI Press/The MIT Press, Cambridge.

Lerman, K. and Galstyan, A. (2002). Two paradigms for the design of artificial collectives. In *First Annual workshop on Collectives and Design of Complex Systems*, CA. NASA-Ames.

Lerman, K., Martinoli, A., and Galstyan, A. (2005). A review of probabilistic macroscopic models for swarm robotic systems. *Swarm Robotics Workshop: State-of-the-art Survey, LNCS 3342*, pages 143–152.

Liu, W., Winfield, A., Sa, J., Chen, J., and Dou, L. (2007). Towards energy optimization: Emergent task allocation in a swarm of foraging robots. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 15(3):289–305.

Mataric, M. (1992). Minimizing complexity in controlling a mobile robot population. In *IEEE International Conference on Robotics and Automation*, pages 830–835, Piscataway. IEEE Press.

Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, New Jersey, USA.

Mondada, F., Pettinaro, G. C., Guignard, A., Kwee, I. W., Floreano, D., Deneubourg, J.-L., Nolfi, S., Gambardella, L. M., and Dorigo, M. (2004). Swarm-bot: A new distributed robotic concept. *Autonomous Robots*, 17(2–3):193–221.

Murphy, R. (2000). *Introduction to AI Robotics*. MIT Press, Cambridge.

Musliner, D., Durfee, E., and Shin, K. (1995). World modeling for dynamic construction of real-time control plans. *Artificial Intelligence*, 74(1):83–127.

Nilsson, N. (1969). A mobile automaton: An application of ai techniques. In *Proc. of the First International Joint Conference on Artificial Intelligence*, pages 509–520. Morgan Kaufmann Publishers, San Francisco, CA.

Nouyan, S., Campo, A., and Dorigo, M. (2008). Path formation in a robot swarm. self-organized strategies to find your way home. *Swarm Intelligence*, 2(1). In press.

Parker, L. (1997). L-ALLIANCE: Task-oriented multi-robot learning in behavior-based systems. *Journal of Advanced Robotics, Special Issue on Selected Papers from IROS '96*, 11(4):305–322.

Parker, L. (1998). ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240.

Parker, L. and Tang, F. (2006). Building multi-robot coalitions through automated task solution synthesis. *Proceedings of the IEEE, special issue on Multi-Robot Systems*, 94(7):1289–1305.

Peshkin, L. (2002). *Reinforcement learning by policy search*. PhD thesis, Providence, RI. Adviser: L. Kaelbling.

Pinciroli, C. (2006). Object retrieval by a swarm of ground based robots driven by aerial robots. Diplôme d'Etudes Approfondies en Sciences Appliquées thesis, IRIDIA, Université Libre de Bruxelles, Belgium.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34.

Saridis, G. (1995). Architectures for intelligent controls. In Bupta, S., editor, *Intelligent Control Systems: Theory and Applications*, Piscataway. IEEE Press.

Siciliano, B. and Khatib, O., editors (2008). *Springer Handbook of Robotics*. Springer, Berlin / Heidelberg.

Siegwart, R. and Nourbakhsh, I. (2004). *Introduction to Autonomous Mobile Robots*. MIT Press, Cambridge.

Soysal, O. and Sahin, E. (2005). Probabilistic aggregation strategies in swarm robotic systems. In *Proc. of the IEEE Swarm Intelligence Symposium*, Pasadena, CA.

Sperati, V., Trianni, V., and Nolfi, S. (2008). Evolving coordinated group behaviours through maximization of mean mutual information. *Swarm Intelligence*, 2:73–95.

Tang, F. and Parker, L. (2005a). Asymtre: Automated synthesis of multi-robot task solutions through software reconfiguration. In *IEEE International Conference on Robotics and Automation*, pages 1513–1520.

Tang, F. and Parker, L. (2005b). Distributed multi-robot coalitions through ASyMTRe-D. In *IEEE International Conference on Intelligent Robots and Systems*, Edmonton, Canada.

Togelius, J. (2004). Evolution of a subsumption architecture neurocontroller. *Journal of Intelligent and Fuzzy Systems*, 15:2004.

Turgut, A., Çelikkanat, H., Gökçe, F., and Şahin, E. (2008). Self-organized flocking with a mobile robot swarm. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 39–46, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The CLARAty architecture for robotic autonomy. In *Proc. of the IEEE Aerospace Conference (Big Sky)*, pages 121–132, Piscataway. IEEE Press.