ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DISI

INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Intelligenza Artificiale

Virtual Sensing Technology applied to a swarm of autonomous robots

Tesi di Laurea di:
MATTIA SALVARO

Relatrice:
Chiar.ma Prof.ssa MICHELA MILANO

Correlatore:
Prof. MAURO BIRATTARI

Anno Accademico 2013/14

Sessione III

Ai miei genitori.

Acknowledgements

I would like to express my gratitude to my supervisors Andreagiovanni Reina and Gianpiero Francesca, for their patient and essential guidance throughout my internship at the IRIDIA lab, and afterwards during the writing of this thesis. Thanks to Dr. Carlo Pinciroli for his important contribution in some key passages of my project. Thanks to Professor Mauro Birattari for wisely coordinating my project activities and granting me access to the resources I needed. Thanks to Professor Michela Milano for introducing me such an interesting discipline and such an advanced laboratory like the IRIDIA lab. Thanks to all the people at the IRIDIA lab for such an intense scientific experience.

I would like to thank also my family: my parents and grandparents that never stop supporting me, morally and economically, during this long and sometimes difficult University period. Thanks to all the old friends and new friends that made this long period so fun, and thanks to the long list of flatmates that made cohabitation a wonderful experience. Finally, a special thanks to the patient person that supported me during my months abroad and took care of me in the writing phase, Manuela Dibenedetto. I would like to share the joy of this achievement with you all.

Thank you,

Mattia.

Abstract

This thesis proposes a novel technology in the field of swarm robotics that allows a swarm of robots to sense a virtual environment through *virtual sensors*. Virtual sensing is a desirable and helpful technology in swarm robotics research activity, because it allows the researchers to efficiently and quickly perform experiments otherwise more expensive and time consuming, or even impossible. In particular, we envision two useful applications for virtual sensing technology. On the one hand, it is possible to prototype and foresee the effects of a new sensor on a robot swarm, before producing it. On the other hand, thanks to this technology it is possible to study the behaviour of robots operating in environments that are not easily reproducible inside a lab for safety reasons or just because physically infeasible.

The use of virtual sensing technology for sensor prototyping aims to foresee the behaviour of the swarm enhanced with new or more powerful sensors, without producing the hardware. Sensor prototyping can be used to tune a new sensor or perform performance comparison tests between alternative types of sensors. This kind of prototyping experiments can be performed through the presented tool, that allows to rapidly develop and test software virtual sensors of different typologies and quality, emulating the behaviour of several hardware real sensors. By investigating on which sensors is better to invest, a researcher can minimize the sensors' production cost while achieving a given swarm performance.

Through augmented reality, it is possible to test the performance of the swarm in a desired virtual environment that cannot be set into the lab for physical, logistic or economical reasons. The virtual environment is sensed by the robots through properly designed virtual sensors. Virtual sensing technology allows a researcher to quickly carry out real robots experiment in challenging scenarios without all the required hardware and environment.

Virtual sensor experiments are hybrid experiments between purely simulated and purely real experiments. Indeed, virtual sensors experiments join the real world dynamics of the real robots to the simulated dynamics of the virtual sensors. The benefits of virtual sensing experiments compared to purely real experiments are presented above. The benefit of virtual sensor experiments compared to purely simulated experiment consists in the fact that the former are one step closer to reality than the latter. Hence, hybrid experiments are closer to reality than purely simulated ones.

The proposed system is composed of a tracking system, a simulator and a swarm of robots. The multi-camera tracking system acquires the position of the robots in the arena. This information is then processed inside a simulator, and the output is delivered to the real robots as virtual sensor values. In Chapter 2, I describe the functioning of the tracking system. In Chapter 3, I present the components of the simulator that realize the virtual sensing environment. In Chapter 4, I illustrate the implementation of three virtual sensors. In Chapter 5, I illustrate the effectiveness of the proposed technology by presenting a simple experiment involving a swarm of 15 robots.

This work led to the writing of an international conference article that has been lately submitted [18]. Moreover, working on the presented technology, I had the chance to collaborate to a set of scientific experiments that resulted in an international conference paper [6] and an international journal article currently under review [5]. Furthermore, I contributed to

the implementation and setup of the tracking system and co-authored the relative technical report which documents its functioning [22].

Sommario

Questo lavoro presenta un'innovativa tecnologia nel campo della robotica degli sciami, o swarm robotics. Il sistema progettato permette ad uno sciame di robot di percepire un ambiente di realtà simulata grazie a dei *sensori virtuali*. La tecnologia dei sensori virtuali offre un'opportunità allettante nell'attività di ricerca in swarm robotics, perché permette ai ricercatori di effettuare in modo veloce ed efficiente esperimenti che altrimenti sarebbero più costosi in termini di tempo e denaro, o addirittura irrealizzabili. Questa tecnologia si dimostra utile in particolare per due applicazioni: da un lato rende possibile prototipare e prevedere gli effetti che avrebbe sul comportamento dello sciame l'aggiunta di un nuovo sensore prima di produrlo; dall'altro permette di studiare il comportamento dello sciame in ambienti che non siano facilmente riproducibili all'interno di un laboratorio, per motivi di sicurezza o anche solo perché fisicamente impossibili.

L'uso dei sensori virtuali per la prototipazione mira a simulare il comportamento dello sciame migliorato con l'aggiunta di sensori nuovi o più affidabili, prima ancora di produrne l'hardware. Prototipare un sensore permette di mettere a punto un nuovo sensore o di confrontare le prestazioni di tipi di sensori alternativi. Questo tipo di studi possono essere condotti con lo strumento presentato in questo lavoro, che permette di sviluppare via software e testare rapidamente sensori virtuali di diverse tipologie e livelli di qualità, simulando il comportamento di sensori reali di varia natura. In alternativa, lo strumento può essere utilizzato per indagare su quale o quali sensori reali sia meglio investire in modo tale da ottenere un dato livello di prestazione dello sciame, minimizzando il costo di produzione.

Con il sistema di realtà aumentata, è possibile testare le prestazioni di uno sciame che opera in un ambiente che non può essere ricostruito in un laboratorio per motivi di natura fisica, logistica o economica. L'ambiante virtuale è percepito dai robot reali attraverso sensori virtuali progettati ad-hoc. Questa tecnologia può essere sfruttata per effettuare velocemente esperimenti con robot reali in scenari innovativi, senza dover predisporre l'ambiente né avere l'hardware necessario a bordo dei robot.

Gli esperimenti con i sensori virtuali caratterizzano un ibrido tra gli esperimenti puramente simulati e quelli puramente reali. Infatti, gli esperimenti con sensori virtuali uniscono le dinamiche reali dei robot a quelle simulate dei sensori virtuali. I benefici degli esperimenti ibridi rispetto a quelli puramente reali sono stati descritti sopra. Il vantaggio degli esperimenti ibridi rispetto a quelli puramente simulati invece consiste nel fatto che i primi sono si avvicinano di più alla realtà rispetto agli ultimi. Di conseguenza anche i loro risultati saranno più congruenti a quelli reali.

Il sistema progettato è composto da un sistema di tracking, un simulatore ed uno sciame di robot. Il sistema di tracking multi camera acquisisce la posizione dei robot nell'arena, ed invia i dati ad un simulatore che li processa. L'output che ne risulta sono i valori dei sensori virtuali, che vengono inviati ai robot reali attraverso una rete Wi-Fi. Nel Capitolo 2 descrivo il funzionamento del sistema di tracking, nel Capitolo 3 presento le componenti del simulatore che realizzano l'ambiente per i sensori virtuali, nel Capitolo 4 mostro l'implementazione di tre sensori virtuali, nel Capitolo 5 illustro l'efficacia della tecnologia proposta con un semplice esperimento con 15 robot reali.

Questo lavoro ha portato alla stesura di un articolo per una conferenza internazionale che è stato presentato di recente. Lavorando su questo progetto, ho anche avuto la possibilità di collaborare ad una serie di esperi-

menti scientifici che sono risultati in un articolo per una conferenza internazionale [6] e un articolo per una rivista scientifica internazionale, al momento in fase di revisione [5]. Inoltre, ho contribuito all'implementazione e messa a punto del sistema di tracking e sono coautore della relativa relazione tecnica che ne documenta il funzionamento [22].

Contents

1 Introduction **3**
- 1.1 Swarm robotics . . . 3
- 1.2 Motivations . . . 5
- 1.3 Overview . . . 6
- 1.4 Virtual Sensing: state of the art . . . 8
- 1.5 Original contribution . . . 9
- 1.6 Thesis structure . . . 10

2 Arena Tracking System **11**
- 2.1 Halcon API Layer . . . 13
- 2.2 Arena Tracking System API Layer . . . 18
  - 2.2.1 Detection and optimisation . . . 19
  - 2.2.2 Configuration . . . 22
- 2.3 Arena Tracking System Application Layer . . . 27
  - 2.3.1 Arena Tracking System Viewer . . . 27
  - 2.3.2 Arena Tracking System Server . . . 28

3 Arena Tracking System Virtual Sensing Plugin **30**
- 3.1 ARGoS overview . . . 30
- 3.2 Virtual sensing with ARGoS . . . 37
- 3.3 Arena Tracking System Virtual Sensing Plugin Simulator Module 40
  - 3.3.1 Arena Tracking System Client . . . 41
  - 3.3.2 Arena Tracking System Physics Engine . . . 45
  - 3.3.3 ARGoS Virtual Sensor Server . . . 47
  - 3.3.4 Virtual Sensors Simulator Module . . . 49
- 3.4 Arena Tracking System Virtual Sensing Plugin E-Puck Module . 51
  - 3.4.1 Virtual Sensor Client . . . 52
  - 3.4.2 Arena Tracking System Real E-Puck . . . 54
  - 3.4.3 Virtual Sensors Real Robot Module . . . 56

4 Virtual Sensors implementation **59**
- 4.1 Control Interface . . . 59
- 4.2 Simulator . . . 61
  - 4.2.1 Ground Virtual Sensor Simulator Module . . . 61
  - 4.2.2 Light Virtual Sensor Simulator Module . . . 64
  - 4.2.3 Pollutant Virtual Sensor Simulator Module . . . 67
- 4.3 Real Robot . . . 71
  - 4.3.1 Virtual Sensor Real Robot Modules implementation . . . 71

5 Validation through real robots experiment **74**

6 Conclusions and future work **77**
6.1 Future work 77

Chapter 1

Introduction

In this thesis, I present a tool to help researchers conducting experiments in the field of swarm robotics. The tool enhances the experimental experience enabling two useful functions: augmented reality for the robots, and robots' sensor prototyping. The project had been entirely developed at the IRIDIA [8] lab, Université Libre de Bruxelles, Belgium, under the supervision of professor Mauro Birattari and Ph.D students Andreagiovanni Reina and Gianpiero Francesca. To better understand the field in which my work settles and its possible applications, I first introduce swarm robotics. Then, I will explain the motivations and goal of the work, followed by an overview of the entire system. Then I summarise the state of the art relevant to the object of my study, that is virtual sensing technology for swarm robotics. At the end of the chapter I illustrate what is my original contribution to this work, and finally I explain how the thesis is structured.

1.1 Swarm robotics

Swarm robotics [4] is a branch of Artificial Intelligence that focuses on the design of self organised groups of autonomous robots that cooperate to achieve a common goal. Swarm robotics assumes absence of any form of centralised control or communication infrastructure. Cooperation and coordination are obtained exclusively through numerous local interactions among the robots. The task of the robot swarm designer is to create individual robot behaviours that allow the swarm to satisfy the global requirements, while exploiting only local interactions among the robots. Robots local interactions can be between robots, or between robots and the environment.

According to the definitions above, the advantageous characteristics of swarm robotics are: fault tolerance, scalability and flexibility. Fault tolerance results from the decentralised behaviour and the high redundancy of the swarm. The failure of one or more single units should not affect the collective behaviour if there is no centralised control, no predefined roles inside the swarm and a sufficient degree of redundancy. Scalability is obtained thanks to local interactions. Global performance is proportional to the number of robots cooperating in the swarm, and no reprogramming of the robots is needed after the swarm size is changed. Flexibility is accomplished by the distributed and self organ-

ised essence of the swarm, that allows the swarm to cope with time-variant environment for instance through dynamic task allocation among the robots.

The potential applications envisioned for swarm robotics are numerous and diverse. In fact, swarm robotics is well suited for tackling tasks such as search and rescue, demining, construction in hostile environment like space or underwater, post-disaster recover. Search and rescue and demining are dangerous tasks for human beings. Employing a fault tolerant swarm of robots for this task is desirable to avoid human losses, because robots losses are tolerated by the system. Thanks to scalability the task execution time can be accelerated by pouring more robots in the swarm. Besides being human threatening, hostile environments usually lacks of any kind of communication infrastructure. Swarm robotics perfectly fits this lack of global communication, because the swarm robotic system relies only on local communication for coordination and cooperation between robots. These low requirements make swarm robotics a viable solution for space or underwater construction. Another human hostile kind of environment is a post-disaster environment: earthquakes, floods, nuclear power plant accidents, wars. Swarm robotics can be employed for sites recovering in these kinds of situations. Assuming for instance a post-earthquake scenario. The tasks to be performed for earthquake recovery are numerous: search and rescue, demolition of unstable buildings, securing of damaged buildings, pipelines, electrics and hydro systems. Thanks to its flexibility, the swarm is able to allocate the tasks to the robots dynamically, adapting the number of robots assigned to one task to the needs of the time variant environment.

According to Brambilla et al. [3], a swarm robotics system can be modelled both at microscopic and macroscopic level. Microscopic models consider the single robots individually, analysing the unit interactions with each other or with the environment. Different levels of abstraction make the model more or less realistic but also more or less complex for design purposes. Macroscopic models take into account the whole swarm as a unique entity. Such models provide a high level perspective of the system and allow a designer to focus on the properties that the swarm requires to achieve.

In reality, requirements are expressed at swarm level, while design must be actualised at microscopic level. Today, a general engineering methodology for swarm robotics design is still missing, and control software for swarm robotic systems is produced in two ways: manual and automatic. In manual design, the gap between the two levels is filled by the individual ability and experience of the designer, making of swarm design a kind of art rather than a strict discipline. The designer follows a trial and error approach, developing the single robot behaviour, testing and improving it until the desired collective behaviour is obtained. The most used software architecture is the probabilistic finite state machine, but another common approach is based on virtual physics, where robots and environment interact through virtual forces. While probabilistic finite state machine is more suited for tasks like aggregation [20], chain formation [15] or task allocation [11, 10], virtual physics is a better approach for spatial organising tasks, for example pattern formation [21] and collective motion [19].

The main automatic design method is called evolutionary robotics [24]. With this technique, single robots are controlled by a neural network whose parameters are obtained through artificial evolution. The main drawback of the automatic design method, is that defining an effective setting is usually a difficult

problem. A novel approach to automatic design, called AutoMoDE, has been proposed lately by Francesca et al. [5].

In the next section I will list the open issues in swarm robotics and the motivations that sustain my work

1.2 Motivations

The application areas mentioned above are only potential. In fact, the complexity of the swarm design and the envisioned applications themselves, prevented swarm robotics to take off in the real world. At date, swarm robotics research is still confined inside research laboratories. Researchers make extensive use of simulation tools to test and evaluate their algorithms, because real robot experiments, even in a highly controlled environment, are still quite an issue. Compared to simulation, real robots experiments are very expensive in terms of time and work, though often necessary. Validation of the control software through real robot experiments is demanded by a typical incongruity between simulated and real robot experiments results. Simulators are unable to model all the aspects of reality, and depending on the level of abstraction, reality is more or less simplified. This simplification, or abstraction of reality, leads to a mismatch between reality and simulation, called reality gap. The presence of the reality gap makes real robots validation necessary in most of the cases, however the execution of real robots experiments is not so trivial and slows down the whole control software production process.

Given the current conditions in swarm robotics research, my work aims to help researchers carrying out their experimental indoor activity and to ease the usage of real robots thanks to an augmented reality environment. I implemented a system that enables virtual sensing on a swarm of robots. Such a system affects the experimental experience in two ways. First, the architecture provides an augmented reality system to allow the creation of a hybrid environment where real robots can sense the simulated environment by means of virtual sensors. Second, the system can be used as a tool to prototype sensors that are currently unavailable on board the robots.

The researcher can exploit augmented reality to create an environment that is not physically replicable inside labs. The inability to replicate the environment can be due to safety reasons, for example dangerous radiations or spreading wildfire, or for practical and economical reasons. For example, the researcher can insert in the environment any source of wave radiation, including light, and implement the corresponding virtual sensor on the robots. As an example, I consider the experiment exposed in [7], where infrared transmitters define different areas in which the space is divided. Let's assume that the robots are equipped with infrared receivers, but the transmitters are not immediately available or must be bought. Virtualising the transmitters and implementing the infrared virtual sensor for the robots allow the researcher to start the experiments right away. Considering the same experiment, the wall of the maze in which the robots operate can be virtualised as well. Walls, objects and obstacles virtualisation gives the researcher high flexibility and simplifies in terms of time and material the set up of the experimental environment.

The system can also be seen under another point of view. In a situation in which the researcher has the chance to upgrade the robots with new pieces of

hardware, virtual sensors can help in prototyping the real sensors before their production. Virtual sensing technology can be exploited for tuning and testing on real robots a particular sensor. Virtual sensors can be easily tuned and tried many times, until satisfactory swarm performances are achieved. Once the sensor features are validated, the hardware sensor can be bought or built once and for all. Prototyping a sensor is particularly useful in swarm robotics because the real sensor must be replicated in several units for all the swarm. Building or buying dozens of sensors can be very expensive, having them tested before is a great advantage. Another application of the virtual sensing system is the virtual enhancement of a sensor that is available on the robots, but for some reasons it is not particularly performing. In this case, the researcher can have an estimate of the performance of the swarm, if robots were equipped with more effective sensors.

The proposed tool represents an innovation in the experimental experience in swarm robotics. After the preliminary set up of the arena, in which the needed infrastructure and hardware must be installed one-off, the tool can be used and extended without further modifications neither in the system infrastructure nor on the robots. The system is extensible to any new virtual sensor and any robotic platform, and both the core and the extensions are all software based. The extensions can be easily implemented by the researcher to add more virtual sensors. In Chapter 4, I guide the reader through the implementation process of a new virtual sensor. In the remainder of the introduction, I give an overview of the system, then I summarise what is the state of the art in virtual sensing technology for swarm robotics and I highlight the innovation brought by the work presented in this thesis.

1.3 Overview

The virtual sensing architecture consists of three components: a tracking system, a robotic swarm simulator and a swarm of robots. Figure 1.1 shows the architecture, its deployment and data flow at high level of abstraction. We can imagine the system as an information elaboration chain, running from the producer to the consumer. The information source is the tracking system, while the final user is the swarm of robots. In between there is an elaboration and communication system that transforms the set of robot positions and orientations produced by the tracking system to a set of virtual sensor readings consumed by the robots.

The cameras of the tracking system acquire the images of the arena where the robotic experiment takes place. The images are then processed by a vision software that outputs positions and orientations of all the robots detected in the environment. The set of robot positions and orientations is called *arena state*. The arena state is the result of the first data elaboration and represents the output produced by the tracking system. The arena state is then transmitted to a robotic swarm simulator that is able to compute the virtual sensor readings for the robots. When the computation is done, the readings are sent to the robots equipped with the corresponding virtual sensors. The result is that the real robots navigating in the arena are immersed in a perceivable virtual environment handled by the simulator. This kind of interaction between simulation and reality is called virtual sensing technology.

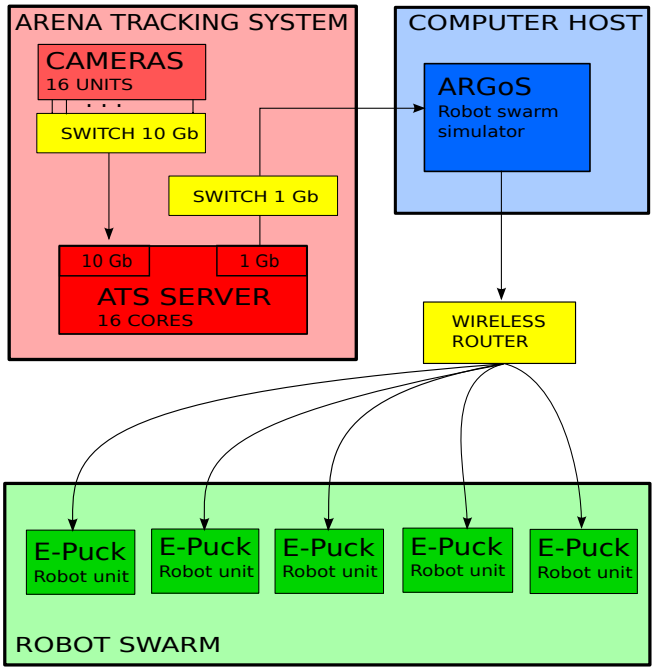

Figure 1.1: Data flow of the system.

The tracking system I used is called Arena Tracking System or ATS [22] and it was built at IRIDIA. The ATS is composed by a set of 16 cameras connected to a dedicated server machine that performs the image processing. The ATS represents the source of the information in the chain elaboration process and even though I used a custom tracking system, any other tracking system can be easily employed, as far it outputs the position and orientation of the robots.

The second element of the elaboration process is the robotic swarm simulator. A robotic swarm simulator is an application that allows the researcher to simulate swarm robotics experiments. There is a number of reason why simulated experiments are used instead or before real robots experiments. The advantages of simulating an experiment mainly affect two factors: time and resources saving. Simulating an experiment can drastically reduce the time duration, and gives the possibility of run experiments in batch without human supervision. Also, simulating does not require resources other than a standard computer. No resources needed means no hardware costs, no hardware failures, and no environment setting. Both robots and environment are set by the researcher in the simulator. One drawback of simulation though is that abstraction is just a simplification of reality, and often simulated results are better than real experiment results.

The role of the simulator in this system is to replicate the real arena state in the simulated environment. Knowing the position and orientation of the robots allows the simulator to calculate the values of the virtual sensors onboard the robots. The robotic swarm simulator I employed was built at IRIDIA and it is called ARGoS [17]. One of ARGoS useful features is modularity. In ARGoS, every component is a modular plugin that can be loaded upon need. I exploited

ARGoS modularity to integrate the virtual sensing system as a plugin module in the simulator. The plugin operates as extension to the simulator core, accessing the core functionalities of the simulator. The virtual sensing plugin includes several basic components: a special physics engine for the simulator, the communication infrastructure needed for connection with both tracking system and robots, and the generic software architecture for virtual sensors. The researcher can extend the system by implementing specific plugins for each virtual sensor needed, extending the generic virtual sensor structure provided by the vitual sensing plugin.

The robotic swarm employed is composed of E-Puck [13] robots. The E-Puck is a desktop mobile robot developed at École Polytechnique Fédérale de Lausanne (EPFL) for educational purposes. The robot set available at IRIDIA lab is a particular extension of the basic E-Puck platform. Among other features, the extension endows the robots of Wi-Fi communication needed to enable the virtual sensing technology.

The result of the interaction between these three macro components of the system is a virtual environment that is perceivable by the real robots, a sort of augmented reality for a swarm of robots. In the next section, I will focus on the current state of the art in virtual sensing applied to swarm robotics, explaining why the work proposed in this thesis brings a novel contribution to this technology.

1.4 Virtual Sensing: state of the art

In literature, the interaction between real robots and virtual environment is discussed in two ways. Some authors proposed or envisioned a virtual sensing technology, others achieved real-robots/virtual-environment interaction through virtual actuation technology. If we think at virtual sensors like data flowing from a simulator to real robots, we can see virtual actuators implemented as information moving from the real robots to the virtual environment simulator. My work does not include a virtual actuation technology, however virtual actuators are a straightforward extension to it. In this thesis, a virtual actuation extension is discussed in Section 6.1.

Millard et al. [12] discuss the importance of a virtual sensor technology, however they envision it only as future work. Instead, O'Dowd et al. [16] and Bjerknes et al. [2] implemented a specific virtual sensor to perform robot localisation. Both works ground their architecture on a tracking or positioning system to import the real robots' global position in a simulator. O'Dowd et al. [16] used a tracking system and Wi-Fi communication to supply the robots with their global position. Thus, virtual sensor technology has been implemented in one specific virtual GPS sensor. Bjerknes et al. [2] developed a 2D and 3D positioning system based on ultrasonic beacons. Thanks to triangulation, the robots are able to calculate their position autonomously. The authors achieved decentralised virtual sensing using an embedded simulator. Although their solution is scalable and does not need Wi-Fi communication, running an embedded simulator on the robots is generally too demanding for hardware architectures used in swarm robotics. Furthermore, the positioning system requires ultrasonic sensors placed ad-hoc on the robots, whereas no need of specific hardware is one of the features of my work. Unlike the works cited above, my work aims to build

a general platform on which any kind of virtual sensor can be implemented, without any other specific hardware installation.

Among those who proposed virtual actuation technology, virtual pheromone is the most studied virtual actuator. In swarm robotics, having virtual pheromone is a great achievement. Pheromone is a means of communication among social insects, and stigmergy, the mechanism of indirect communication between social insects, is a very hot topic in swarm robotics. Therefore, to have the possibility to test stigmergic algorithms based on pheromone in a real robot experiment is a huge advantage. Sugawara et al. [23] and Garnier et al. [7] simulated pheromone trails deployment using coloured light projections on the floor. Their implementation of pheromone requires special hardware on the robots and a smart environment in which the control of the light is limited by the usage of a projector. Khaliq et al. [9] studied stigmergic behaviour in swarm robotics abstracting the properties of pheromone and implementing them with a grid of radio frequency identification (RFID) tags. The RFID tags are hexagonal cells embedded in the floor. They have an unique identifier and a readable and writable memory. The robots are equipped with RFID transceiver and they are able to identify the tag below them and read or write the value stored in memory. With this system, Khaliq et al. succeeded in virtualising pheromone for stigmergy studies, i.e. to virtualise a particular actuator. The system they propose is scalable because the information is externalised and spatialised on the RFID tags. The robots only elaborate local information and there is no need of central control and computation. However, all the works cited above require a smart environment and special hardware on the robots.

The extension to my work discussed in Section 6.1 will provide virtual actuation technology totally software implemented. Through this extension it will be possible to carry out the foraging task based on virtual pheromone with real robots, as simply as running a simulation.

1.5 Original contribution

The system described in Section 1.3 is extended and articulated. It is composed of three subsystems in an organic interconnection that exploits three different networks and several hardware resources. The three subsystems, ATS, ARGoS and the E-Puck swarm, were originally available at IRIDIA, however no communication protocol was there because they were not supposed to interact with each other at the time. In addition, the ATS needed to be tuned and integrated with some missing features. In the first part of my work, I fine tuned the tracking system and developed two utility applications. The fine tuning of the tracking system consists in the calibration of the 16 cameras, the creation of a performing tag pattern to achieve better robot detection, and the optimisation of the image processing algorithm. The applications I developed are a tool for real time view of the tracking session called Viewer, and a networking component playing as a server to provide arena state data to the simulator, called Arena Tracking System Server. This tool has been used for a set of scientific experiments allowing me to collaborate for an international conference paper [6] and an international journal article under review [5]. Furthermore, I contributed to the implementation and setup if the ATS and I co-authored the technical report [22].

The second part of my work focused on the integration of ARGoS and the ATS, and the creation of the virtual sensor architecture. I designed and implemented a special physics engine to be plugged in ARGoS, equipped with the necessary networking components to handle the connections with the Arena Tracking System Server and the entire set of robots. Contextually, I designed and implemented a generic virtual sensor architecture that consumes the data generated by the upstream system, and works as a frame structure for virtual sensor implementation.

Finally, I illustrate the functioning of the system through the implementation of three virtual sensors: the ground virtual sensor, the light virtual sensor and the pollutant virtual sensor. I tested their effectiveness in a simple experiment described in Chapter 5. This work led to the writing of an international conference article that has been lately submitted [18].

1.6 Thesis structure

The thesis is structured as follows: Chapter 2 describes the architecture and the features of the ATS, Chapter 3 focuses on the design of the virtual sensing plugin for ARGoS, Chapter 4 offers the guidelines for virtual sensor implementation, Chapter 5 defines and execute a real robot experiment to test virtual sensor effectiveness, and finally Chapter 6 resumes the conclusions and suggests the possible future works.

Chapter 2

Arena Tracking System

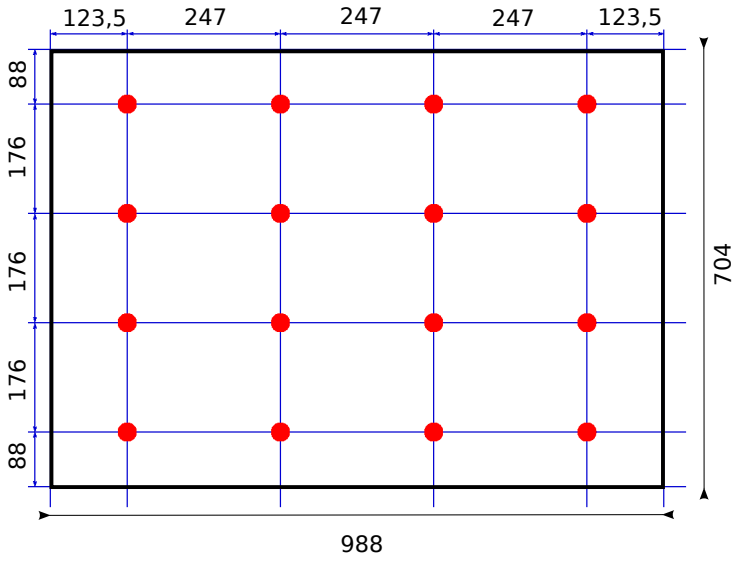

Figure 2.1: Camera placement in the arena (distances in cm).

The Arena Tracking System is composed by a set of 16 Prosilica GC1600 C cameras, manufactured by Allied Vision Technology. The cameras are installed inside the IRIDIA Robotics Arena, in which they are disposed on a framework suspended 243 cm above the floor in a 4 by 4 matrix formation (see Figure 2.1). The total scope of the 16 cameras is nearly $70m^2$, which makes the ATS coverage area more than enough for any swarm robotics experiment run in IRIDIA. The cameras resolution is 1620 x 1220 pixel and the maximum frame rate is 15 frames per second. However the output rate of the tracking system is lower due to the time demanding elaboration process performed on each set of images acquired.

The image processing is made on a dedicated server machine that hosts 16 Intel®Xeon®CPU E5-2670 at 2.60GHz. Each core features the Intel®Hyper-Threading Technology, which enables it to run two threads per core. The Operating System is GNU/Linux Ubuntu 12.04.1 LTS for 64 bits architectures. The cameras are connected to the server through a dedicated switch installed in the

arena. The switch links the cameras with 1 Gb Ethernet connection and the server with a 10 Gb Ethernet connection.

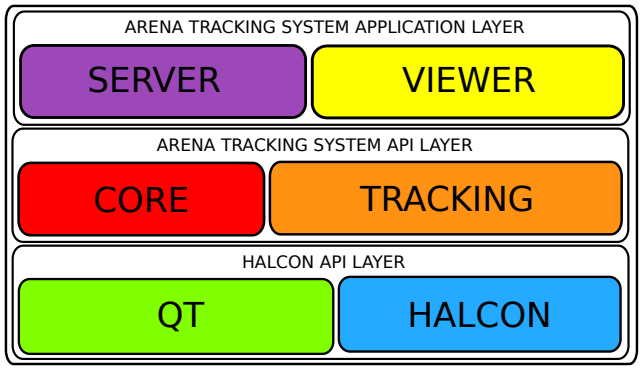

Figure 2.2: Arena Tracking System Software Architecture.

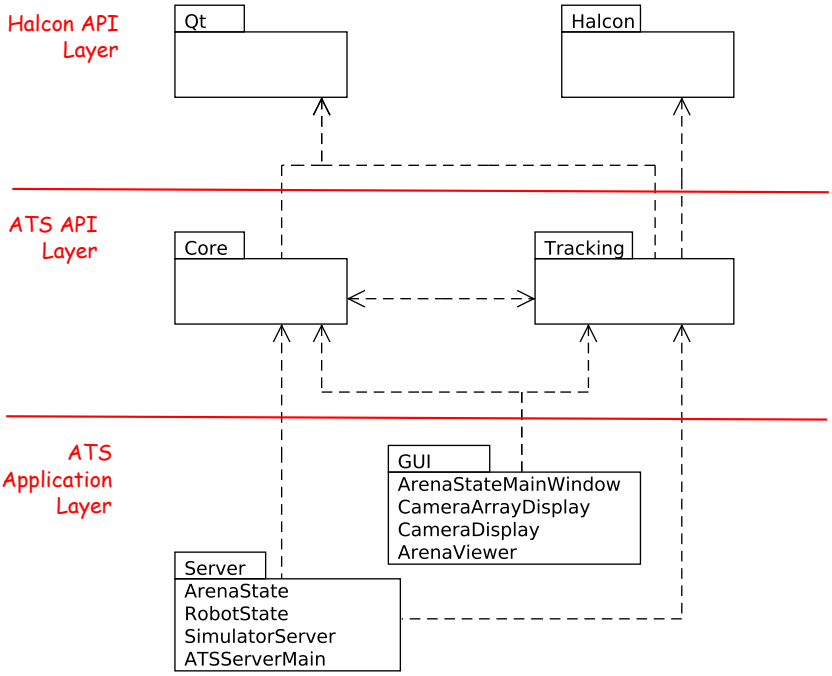

Figure 2.3: Arena Tracking System package diagram.

The Arena Tracking System software architecture is built as a three layer structure presented in Figure 2.2. The bottom layer consists on the QT Framework and the Halcon machine vision library [14]. For further details on QT Framework and Halcon library see Section 2.1. On top of those libraries lies an application specific library layer, called Arena Tracking System API. This custom library offers to the user the possibility to configure the tracking process, as shown in Section 2.2. The top layer is the ATS Application layer. It provides two tools to support researchers' experiments: a viewer for ATS standalone us-

age, and a server for ARGoS integration. The applications receive as input two parameters: a resources XML file and a configuration XML file. The resource file indexes all the resources available for a tracking session such as cameras, image grabbers and image trackers. The configuration file is a subset list of the available resources that the researcher wants to employ in the experiment. The configuration file allows the researcher to nimbly change the tracking system's configuration parameters without affecting the hardware or the compiled code. The applications and the XML files are described in detail in Section 2.3. Figure 2.3 shows the software packages and their dependencies.

In the rest of this chapter I examine the Arena Tracking System from a software engineering perspective, adding technical information for profitable usage.

2.1 Halcon API Layer

The tracking system basic API layer is composed of QT and Halcon libraries. QT is a cross platform application library particularly useful for GUI development thanks to the signal/slot mechanism. Halcon is a library produced by MVTec Software GmbH used for scientific and industrial computer vision applications. Halcon library provides a broad suite of image processing operators, and comes with interactive IDE called HDevelop and drivers to interface a large set of cameras. HDevelop comes very handy for camera calibration and configuration, and allows fast program prototyping through an interactive environment.

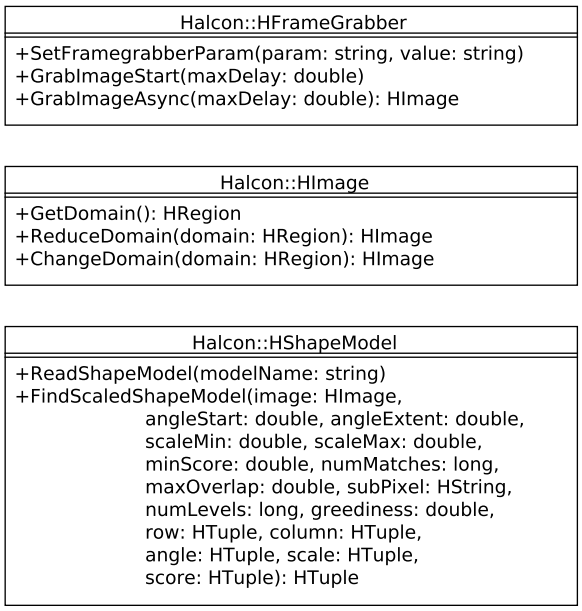

Figure 2.4: Halcon classes used in ATS.

The Halcon library for C++ consists of a set of classes for object oriented programming and a list of operators accessible as static Halcon class methods. The most important Halcon classes for the Arena Tracking System are

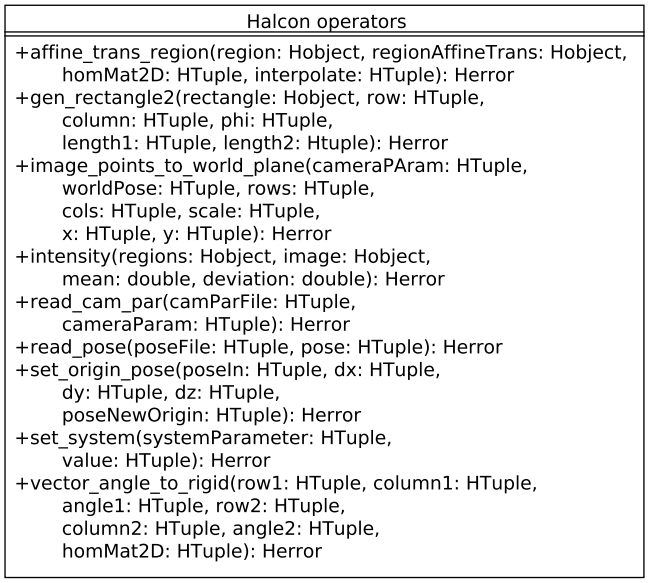

Figure 2.5: Halcon image processing operators used in ATS.

HFrameGrabber, HImage and HShapeModel (see Figure 2.4). HFrameGrabber models an instance of the image acquisition device. The HFrameGrabber class gives the possibility to set specific parameters of the frame grabber and to acquire images either in a synchronous or asynchronous way. In the synchronous mode, the grabber and the image process run sequentially, in such a way that the next grabbing is performed only when the previous processing is done. The time required for processing is therefore included in the frame rate. The asynchronous mode instead allows image processing while the next image is already being grabbed. Due to the high time consuming image process, Arena Tracking System uses asynchronous grabbing to enhance the frame rate.

HImage represents the instance of an image object. HImage object and its methods are used for the particular purpose to reduce the image domain to be processed. Reducing the image domain means to speed up the tag detection, and therefore to raise the efficiency of the tracking system (see Section 2.2.1).

HShapeModel models the instance of a shape model for matching. The shape model is created using HDevelop and saved as a configuration file. The class allows the HShapeModel object to load the shape model file (operator *ReadShapeModel()*), and to find the shape model within an image (operator *FindScaledModel()*), given some parameters such as: rotation range, scale range, the maximum percentage of occlusion of the target, the maximum number of matches to be found, the maximum degree of overlapping between two targets to consider them the same instance, sub pixel accuracy mode, the number of pyramid levels of the shape model used during the search, and a parameter called greediness that is a trade-off between efficiency and accuracy. The operator *FindScaledModel()* outputs the number of targets detected, and the position and orientation of each of them within the image.

Halcon also provides a broad range of operators that are not related to any particular class, but available in C++ as static methods of the generic class

Halcon (see Figure 2.5). I am going to give a brief explanation of the most used operators, mainly used to decode the inner region of the E-Puck tags such as: *vector_angle_to_rigid()*, *gen_rectangle2()*, *affine_trans_region()*, and *intensity()*. The operator *vector_angle_to_rigid()* computes and returns, in terms of rotation matrix and translation vector, a rigid affine transformation between two given points and corresponding orientations. The operator *affine_trans_region()* applies an affine transformation to the given region and returns the transformed region. The operator *intensity()* calculates the mean and deviation of grey values within the input region. The operator *gen_rectangle2()* simply creates an oriented rectangle.

The rest of the operators are used for camera setting or camera view related operation. They are: *read_cam_par()*, *read_pose()*, *set_origin_pose()*, and *image_points_to_world_plane()*. The operators *read_cam_par()* and *read_pose()* read from file the internal camera parameters and the camera pose respectively. The operator *set_origin_pose()* translates the origin of the 3D pose by a given vector. It is used to correct the real world coordinates of the robots, that are originally imprecise due to a perspective error. In fact, while the height of the cameras in the camera pose is the distance between the camera and the floor, the markers created for detection are placed on top robots. To compensate this error it is necessary to translate the camera pose on the z-axis for the height of the robots. The operator *image_points_to_world_plane()* transform a position in the image to a position in a real world coordinate system. This is used to have the position of the robots inside the arena. One last generic operator is *set_system()* and allows to change many Halcon parameters.

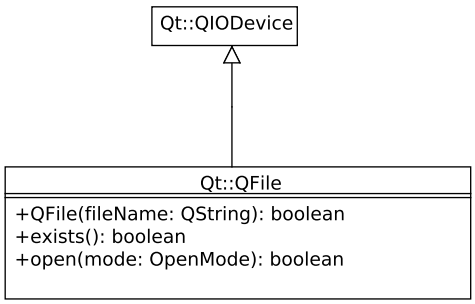

Figure 2.6: QFile class diagram.

Although the ATS API Layer does not need any graphical library, it exploits the QT library for parsing the XML configuration file. The QT classes used for XML file parsing are QFile, QIODevice, QDomDocument, QDomElement and QString. QFile acquires a file given the file name (See Figure 2.6). QT allows to call some utility methods on a QFile object such as *exists()*, and *open()*. QIODevice class provides a public flag type called OpenMode that lists all the possible modes for file opening. QDomDocument class extends QDomNode with the ability to set the content of the file to the QDomDocument object (see Figure 2.7). QDomElement inherits as well from QDomNode class and represents a tag of the XML file. QString is simply a QT wrapper for the string type.

The QT library is used for graphical purpose in the application layer. In particular, the GUI package exploits a set of QT classes to build the user in-

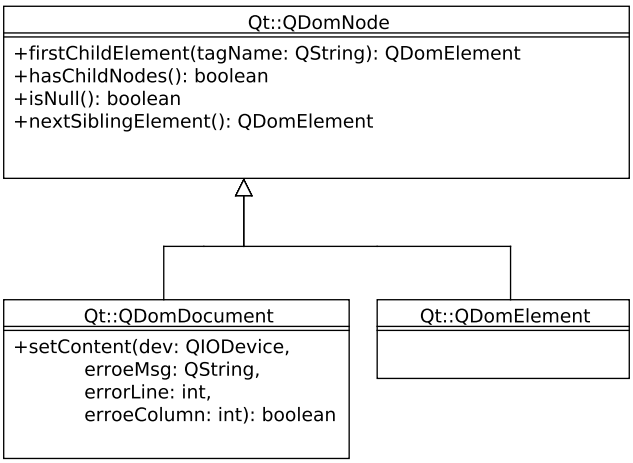

Figure 2.7: QDom classes diagram.

terface of the viewer. The basic component of the QT GUI framework is the widget, and all the GUI components inherit from the class QWidget. The application main window itself extends QWidget class. The arena viewer main window inherits form QMainWindow to access all the QT window framework features such as menus, toolbars, labels and text areas. Figure 2.8 shows the QT classes used for the arena viewer and their most useful methods.

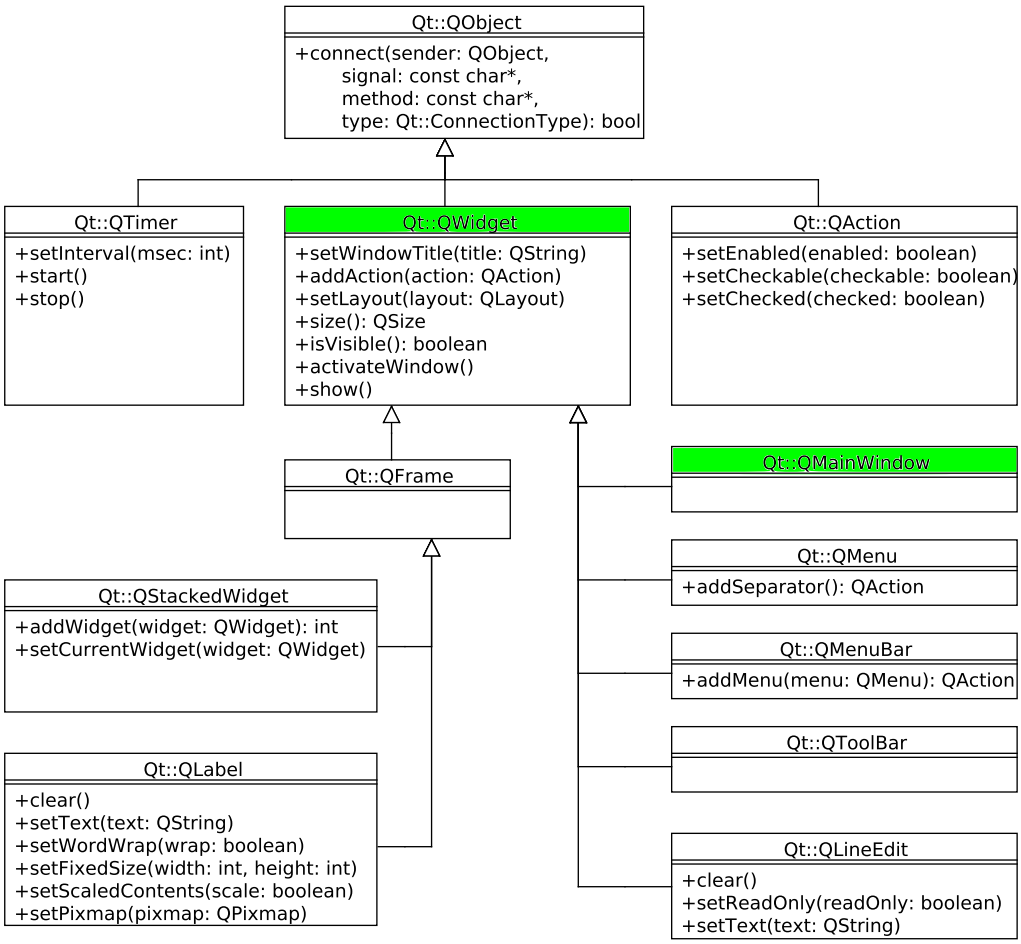

Figure 2.8: QT GUI elements class diagram. The principal classes QWidget and QMainWindow are highlighted.

2.2 Arena Tracking System API Layer

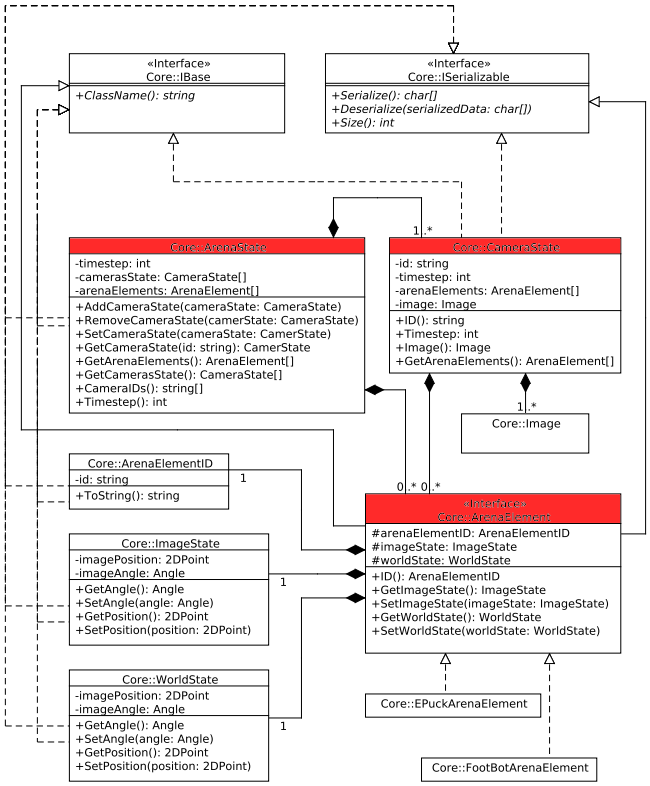

Figure 2.9: Core package class diagram. The principal classes are highlighted.

As stated above, the Arena Tracking System API layer provides a set of functionalities tailored for tracking purpose. The layer is divided into two packages called Core and Tracking (see Figure 2.3). The package Core contains all the basic data structures and provides all the needed interfaces to access them. The data types include the detected elements, the state of a single camera scope, the state of the entire arena space, the representation of the acquired images, plus a resource manager class that parses an XML configuration file and creates the specific image grabber, tracker, and camera (see Figure 2.9 and 2.10). The main class in the Core package is ArenaState, that represents the state of the experiment at a given timestep, i.e. the position and orientation, in the image and in the real world, of every detected robot in the arena. The detected robot is modelled by the class ArenaElement, and it is represented by its own robot ID, its position and orientation in the image and in the real world. The list of ArenaElements is given by ArenaState with the method GetArenaElements(). Alternatively, ATS offers the possibility to get the detected robots under a specific camera, by requesting access to the camera with the method GetCameraState().

The package Tracking includes the single camera tracker, the entire arena tracker, and the interfaces apt to access several image grabbers, trackers and camera views, generated at runtime through a specific factory (see Figure 2.11).

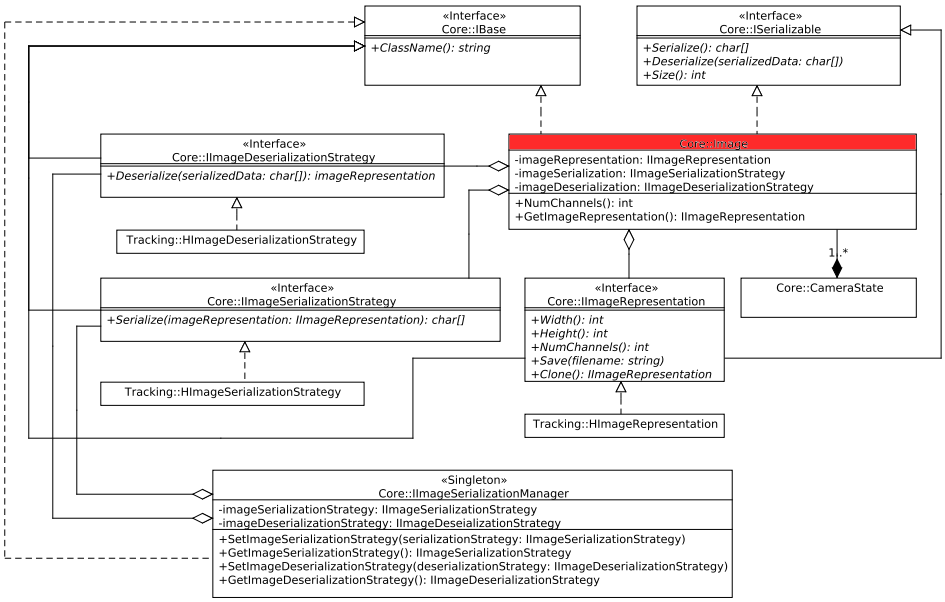

Figure 2.10: Core package class diagram. The principal classes are highlighted.

The class ArenaStateTracker is able to create a new updated instance of ArenaState with the method GetArenaState(). ArenaStateTracker is a manager for one or more CamerStateTracker, to which the GetArenaState() signal is propagated with GetCameraState(). Each CameraStateTracker is associated to an image source and the operators to detect the robots in the image. The CameraStateTracker is composed by an ImageGrabber that encapsulates the logic to configure an image source and acquire an image, an ImageTracker that is responsible for detecting the robots in the acquired image, and a CameraView whose job is to convert the image position of the detected robots into the coordinates of a world reference system, given the internal and external parameters of the specific camera.

2.2.1 Detection and optimisation

As previously mentioned, the instance of ArenaStateTracker applies detection and decoding operators on the input image at every timestep. In order to spot and identify the robots navigating in the arena, the Arena Tracking System uses a type of marker that allows easy detection and decoding even when the image is relatively small. The markers encode the direction of navigation and the ID of the robot at once. Figure 2.12 shows an example of E-Puck marker. The external black arrow indicates the navigation direction, whereas the internal square matrix encodes the robot ID in binary notation, where black squares encode ”0” and white squares encode ”1”. The rising power order of the bits goes from left to right, from top to bottom. Each time the detection step is performed, every frame acquired by the ArenaStateTracker is scanned for occurrences of the marker. Each time a marker is detected, the inner matrix is decoded and converted to an ID.

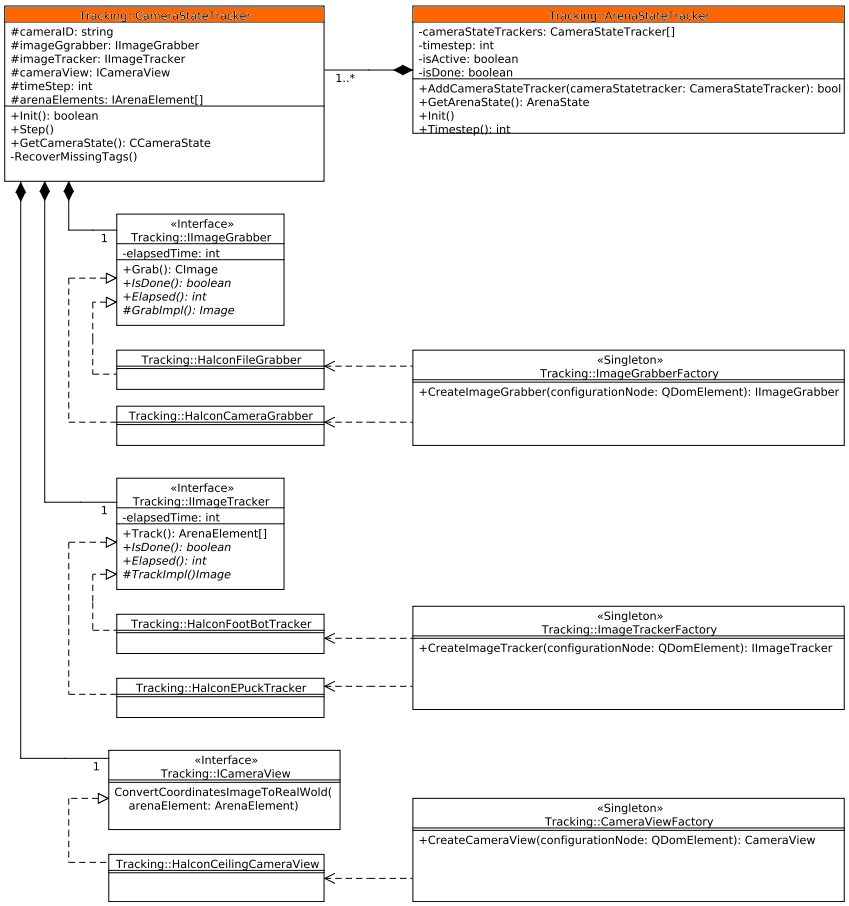

Figure 2.11: Tracking package class diagram. The principal classes are highlighted.

Markers detection is a very time consuming operation due to the high resolution of the cameras mounted in the arena (i.e., 1620×1220 pixels). Although the dedicated machine on which the Arena Tracking System is installed consists of 16 cores, and considered that Halcon operators can be parallelised on those cores, the marker detection on a single image takes on average over 100 ms. In multi camera experiments, the total average tracking time for one frame is the detection time multiplied by the number of cameras used. In a word, average tracking time breaks the requirement of good time resolution settled by the necessity to have real time virtual sensor data and seamless ARGoS integration. It had been empirically proven that 100 ms is a good timestep period for ARGoS. To achieve effectiveness in the integration with ARGoS, the average tracking time per timestep must be around 100 ms. Which is true only for the single camera scenario. Therefore, optimisation in the image processing is needed. Scaling the images is not a viable option because the tiny squares of the markers are represented by only four pixels in the full definition image. A more appropriate solution is to implement an optimisation heuristic that processes the whole image, or set of images, only at a certain periodic timestep, called

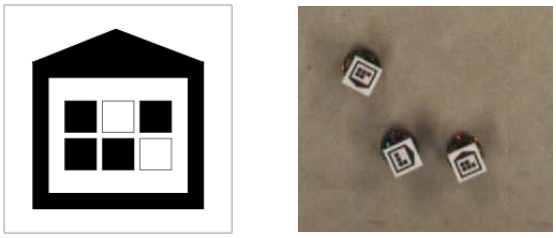

Figure 2.12: On the left: example of E-Puck marker. On the right: markers applied on E-Pucks during an experiment.

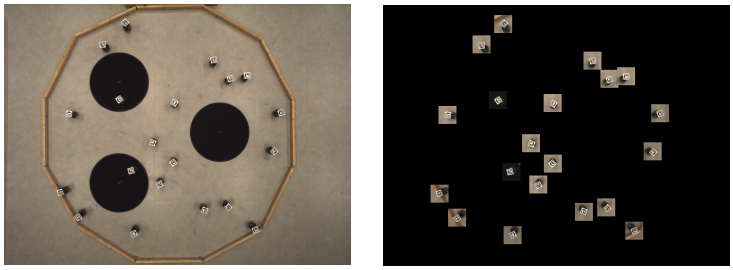

Figure 2.13: On the left: the image domain processed in a keyframe. On the right: the image domain processed in a optimised frame.

keyframe. In the timesteps between two keyframes, the detection is performed only in the neighbourhood of a previously detected marker (see Figure 2.13). The size of the neighbourhood and the keyframe period are parameters that the researcher is able to set in the XML experiment configuration file.

This technique is consistent under the hypothesis that the robots move at a given maximum speed and therefore the maximum distance covered in 100 ms is easily computable. The periodic keyframe helps recovering the robots that may be lost by the ATS during the experiment. In fact, experimental data demonstrate that the reliability of the ATS on markers detection is constrained to keyframe period and neighbourhood size. Table 2.1 shows the number of detected robots for different configuration of the parameters keyframe period and neighbourhood size.

ATS marker detection reliability			
Keyframe period (timesteps)	Neighbourhood size (cm)	Mean	Median
1	–	19.25725	20
5	30	19.05596	19
5	11	19.06481	19
5	10	19.02939	19
9	11	19.01369	19

Table 2.1: ATS marker detection reliability in an experiment with a set of 20 robots.

The experiments are carried out with a set of 20 robots. Table 2.1 and Figure 2.15 show that the system is more reliable for lower keyframe period and higher neighbourhood size. Figure 2.14 demonstrate that lowering the keyframe period and raising the neighbourhood size negatively affects efficiency. The researcher is responsible of settling the trade-off between time efficiency and detection reliability acting on the two parameters in the XML experiment configuration file.

2.2.2 Configuration

Two other important classes of the ATS API layer are ResouceManager and ExperimentManager (see Figure 2.16 and 2.17). ResourceManager is responsible for declaring the set of resources that a researcher wishes to employ for image acquisition and marker detection, while ExperimentManger is delegated for configuring the tracking session. Instances of these classes are meant to load an XML configuration file and to create the instances of the appropriate resources. In particular, the ResouceManager loads and instantiate the contents of the resources XML configuration file, the ExperimentManager creates the objects described in the experiment XML configuration file. The resources XML file contains the definition and the parameters of cameras, detectors and image transformations available for the session. Each of these elements are mapped to a unique ID, that can be used in the configuration file to refer the resource.

The XML tree of the resource configuration file contains a root node, called *arena_tracking_system*, and three children nodes: grabbers, trackers, and cameras. The grabbers node contains the definition of every possible image source available in the experiment. It can be either a camera or directory of images. For each grabber of type camera, in the cameras section must be added a node called camera that contains the path to the camera calibration files. Also, for each grabber of type camera a set of parameters is defined. The camera parameters in the resource configuration files are particularly useful for the researcher to change the brightness of the acquired images. In fact, quite often the brightness of the environment is constrained to some experiment requirement. For example, when the experiment requires a special light source or the coloured LEDs of the robots, the light of the environment must be dimmed to avoid interference. A semi dark environment prevents the tracking system to detect and decode correctly all the markers. Acting on the camera parameters allows the acquisition of clearer images and therefore a more efficient markers detection and decoding. The parameters involved with the brightness of the image are *exposure_time_abs* and *gain_raw*. Raising the gain leads to brighter but more noisy images. In turn, exposure time increases proportionally to the risk to have blurred markers due to the movement of the robots. In the trackers section all the available trackers are listed in nodes called tracker. A parameter that the researcher might find useful here is *robot_height*. This parameter allows the researcher to reuse the same set of markers on robots of different heigh, provided that in the same experiment the set of robots is homogeneous.

The ExperimentManager is instead used to read the experiment XML configuration file that describes the configuration of the ArenaStateTracker in terms of its components. To configure an instance of ArenaStateTracker, the researcher only needs to specify in the XML file how many CameraStateTracker are part of the ArenaStateTracker and, for each of them, specify the resources they should

use calling them by their unique ID.

An example of XML tree for an experiment configuration file includes a root node called *arena_tracking*, and two children nodes: *arena_state_tracker* and *experiment_record*. The *arena_state_tracker* node includes three attributes and a collection of nodes called *camera_state_tracker*, one for each grabber employed in the experiment. The *camera_state_tracker* must include the specification of the grabber, a corresponding camera view if the grabber is of type camera, and the tracker. Back to the *arena_state_tracker* node, the attribute *server_port* defines the port to which the ATS Server is bound, in case the researcher wants to use the ATS this way (see section 2.3.2). The attributes *opt_key_frame_period* and *opt_square_size* are optimisation parameters (see section 2.2.1). The former defines the period in timesteps of the key frame occurrence, the latter is the dimension in centimetres of the side of the square that forms the neighbourhood around the formerly detected markers, on which the image domain is cropped.

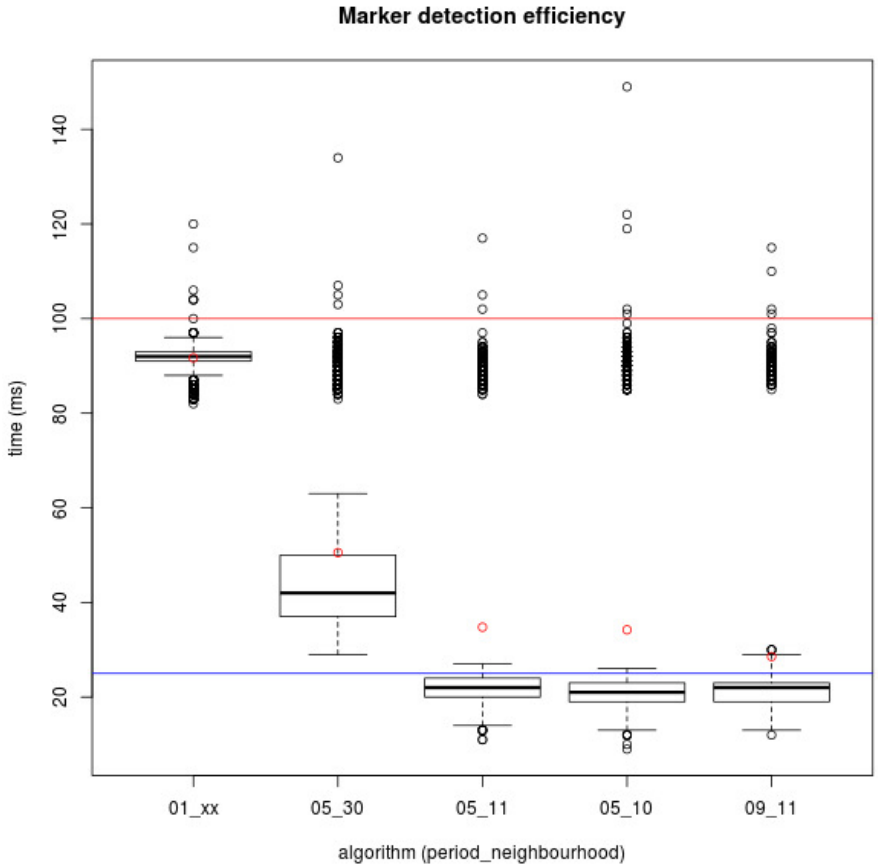

Figure 2.14: ATS marker detection efficiency in ms. The x-axis values encode the keyframe period an the neighbourhood size divided by an underscore. The y-axis shows the time of each timestep in ms. The red line at 100 ms is a required upper bound for efficiency performances. The blue line at 25 ms is an ideal upper bound that allows a four camera experiment to meet the requirement of 100 ms per timestep. The red points represent the average computation time for each combination of keyframe period and neighbourhood size.

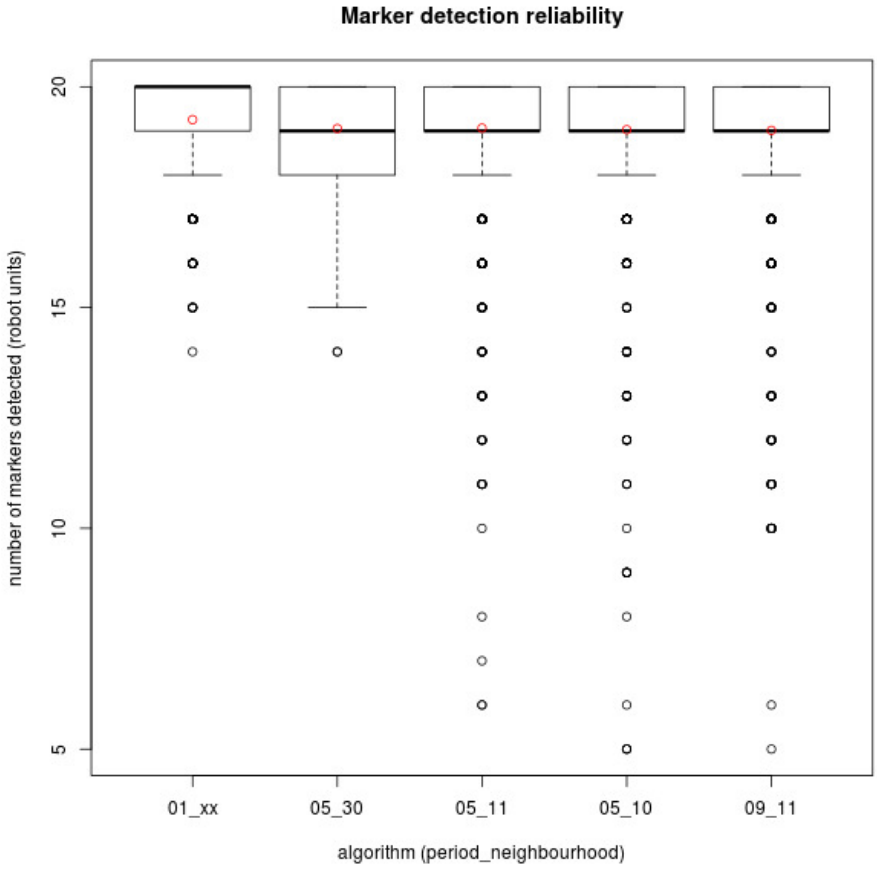

Figure 2.15: ATS marker detection reliability on a set of 20 robots

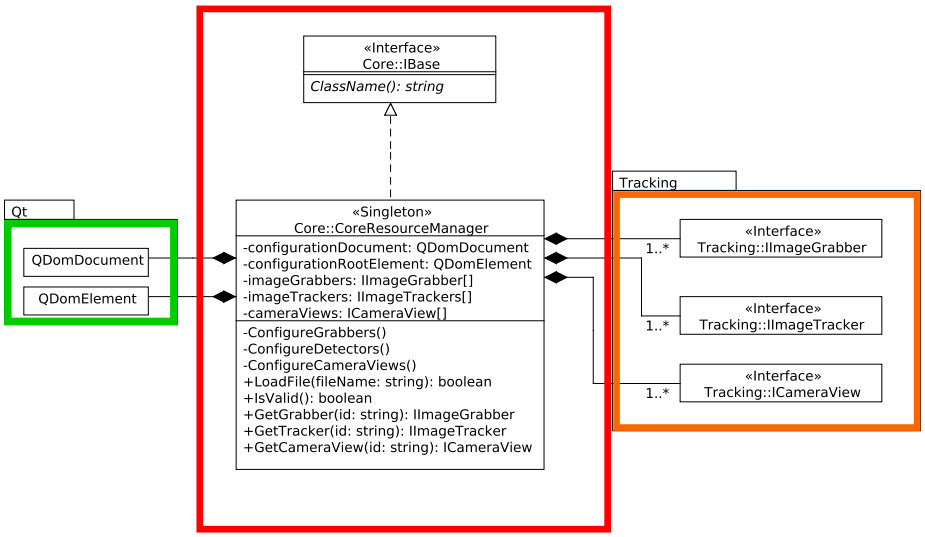

Figure 2.16: ResourceManager class diagram.

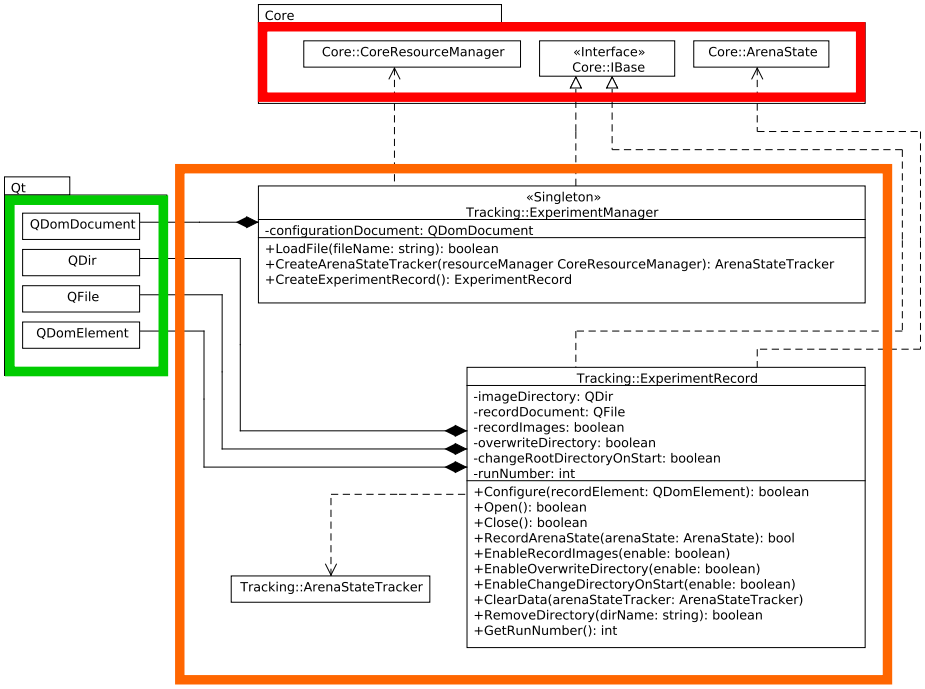

Figure 2.17: ExperimentManager class diagram.

2.3 Arena Tracking System Application Layer

As stated earlier, there are two ways to use the ATS. One is to simply monitor and record the real time evolution of the experiment on all the cameras involved, manually controlling the beginning and the end of the video. For this purpose, an application called Viewer has been created. The Viewer usage do not involve any simulator of virtual sensing technology, it is only a tool to help carrying out traditional robotic experiments. The second provided usage hits the focus of the thesis and enables virtual sensing. This usage allows the researcher to monitor and control the execution of the experiment through the robotic swarm simulator ARGoS, opening a wide range of potential application already mentioned. For this purpose, the ATS Server has been implemented to communicate with a client built ad-hoc within ARGoS (see section 3.3.1).

In this section, the two applications will be presented in detail.

2.3.1 Arena Tracking System Viewer

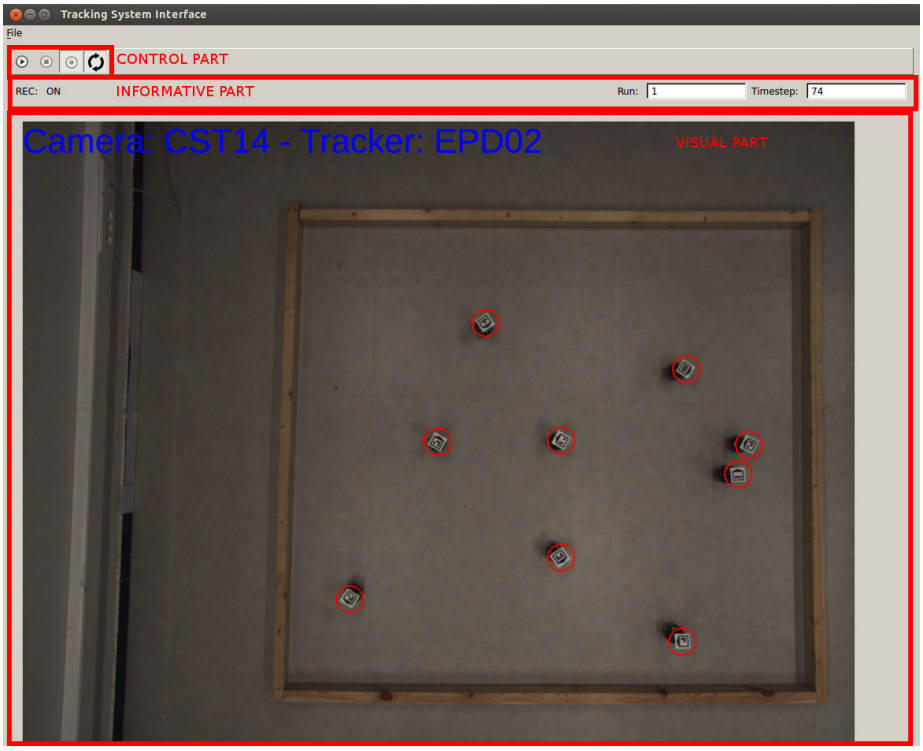

Figure 2.18: Screenshot of the Viewer's GUI.

The Arena Tracking System Viewer (ATS-V) is an application that allows a researcher to visualise and record the progress of the experiment through the images of the active cameras. The ATS-V needs a resource file and a configuration file as argument to set up the desired view. The tool supports any possible subset of cameras among those specified in the resources file, and lets the researcher start and stop the visualisation of the images. The ATS-V also

allows to store the frames and record the whole experiment for further elaboration. For example, the researcher may like making a video of the experiment or tracking the robots offline, using a file grabber instead of a camera grabber. The recording of the images must be enabled in the experiment configuration file in order to be effective in the ATS-V application.

The viewer's GUI is composed of three main parts: the visual panel, the control panel, and the informative panel (see Figure 2.18). The visual panel is a window in which all the cameras stream is displayed according to their position in the camera grid in the arena. To better recognise the camera in the grid, each view reports its camera ID and tracker ID. The control panel disposes the buttons that the researcher can use to start and stop the tracking session. The recording button is shown only for completeness, but its status is bound to the corresponding attribute *record_image* in the experiment XML configuration file. The refresh of the visual panel heavily affects the tracking performance. The refresh toggle button allows the researcher to enable or disable the image refresh in order to gain in terms of performance.

The informative panel provides information about the status of the experiment, such as enabled or disabled image recording, current experiment run number and current timestep. The function associated to the run button can be set in the experiment configuration file with the attribute *change_root_directory_on_start*. If the attribute is set to true, each run is a session between two consecutive start and stop. Otherwise, the run will be only one, but with the possibility to suspend and resume the tracking, and therefore the recording if enabled.

2.3.2 Arena Tracking System Server

The Arena Tracking System Server (ATS-S) application is the interface between the ATS and ARGoS. The application must be launched with the path to a resources configuration file and an experiment configuration file as arguments. ATS-S binds a TCP socket on the two network interfaces of the ATS machine to the port specified in the attribute *server_port* in the experiment XML configuration file. The client within ARGoS must know the address and port on which the ATS-S is waiting to open a TCP connection (see Figure 2.19). After binding, the server enters its main loop. The loop consists of two simple steps: waiting for a connection and enter an inner loop that repeatedly executes commands sent by the client. The inner loop ends when the client disconnects, then the control returns to the main loop that start again waiting for a new connection. The commands that the ATS-S is able to detect and execute are three: *start_experiment*, *stop_experiment* and *one_shot*. The first two commands begin and conclude respectively a tracking session, while the one_shot command performs tracking only on one frame and returns the arena state only once, without incrementing the timestep counter. The command one_shot is particularly useful at the beginning of the experiment to initialise the arena state data structure before actually starting the experiment.

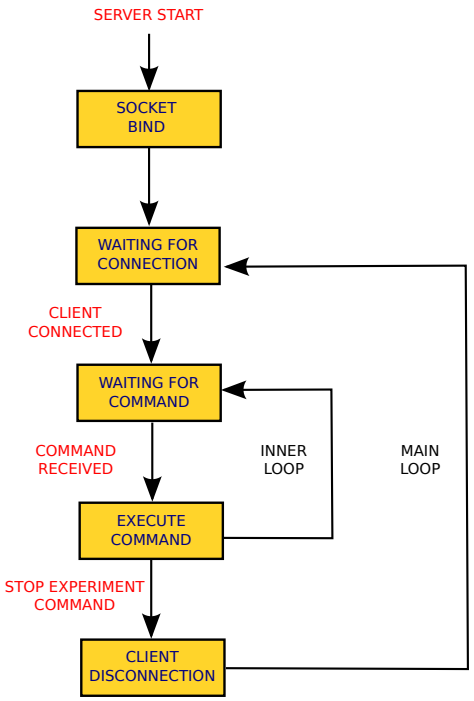

Figure 2.19: Workflow of the ATS-S application.

Chapter 3

Arena Tracking System Virtual Sensing Plugin

This chapter discusses the Arena Tracking System Virtual Sensing Plugin, or ATS-VSP, the tool that implements all the features needed to provide virtual sensing technology to a swarm of E-Pucks. Among other components, the ATS-VSP contains the Arena Tracking System Physics Engine, or ATS-PE, the core of the virtual sensor design, and also the link between simulation and reality.

The ATS-VSP works within ARGoS as a physics engine plugin. To better understand what a physics engine really is in ARGoS, I am going to give the reader a quick overview of the robotic swarm simulator, why I used it, and what are the advantages of integrating the Arena Tracking System in it. The other components included in the ATS-VSP are the networking components for communication with ATS-C and E-Puck swarm, and the virtual sensor bare bones architecture. These components will be described thoroughly in Section 3.3 and Sensor 3.4, covering the simulator and real robot module respectively.

3.1 ARGoS overview

ARGoS [17] is a modular, multi-threaded, multi-engine simulator for multi-robot systems developed at the IRIDIA lab. ARGoS achieves both efficiency and flexibility through parallel design and modularity.

In ARGoS, all the main architectural components are designed as selectable modules, or plugins. The advantage of modularity is that each selected module is loaded at runtime, allocating the resources for the interested aspects of simulation and neglecting unneeded computation. Modularity also eases flexibility. In fact, the plugin structure allows the researcher to modify, diversify and create modules. In ARGoS, all kind of resources are plugins, including robot control software, sensors, actuators, physics engines, visualisations and robots. Several implementations of the same plugin are possible, and usually they differ for accuracy and computational cost. Thanks to this kind of tunable accuracy, the researcher is able to choose the precision of any aspect of the simulation, improving efficiency and flexibility at once.

To clarify the modular architecture of ARGoS, let's exemplify the simulated 3D space. In ARGoS, the simulated 3D space is a collection of data structures,

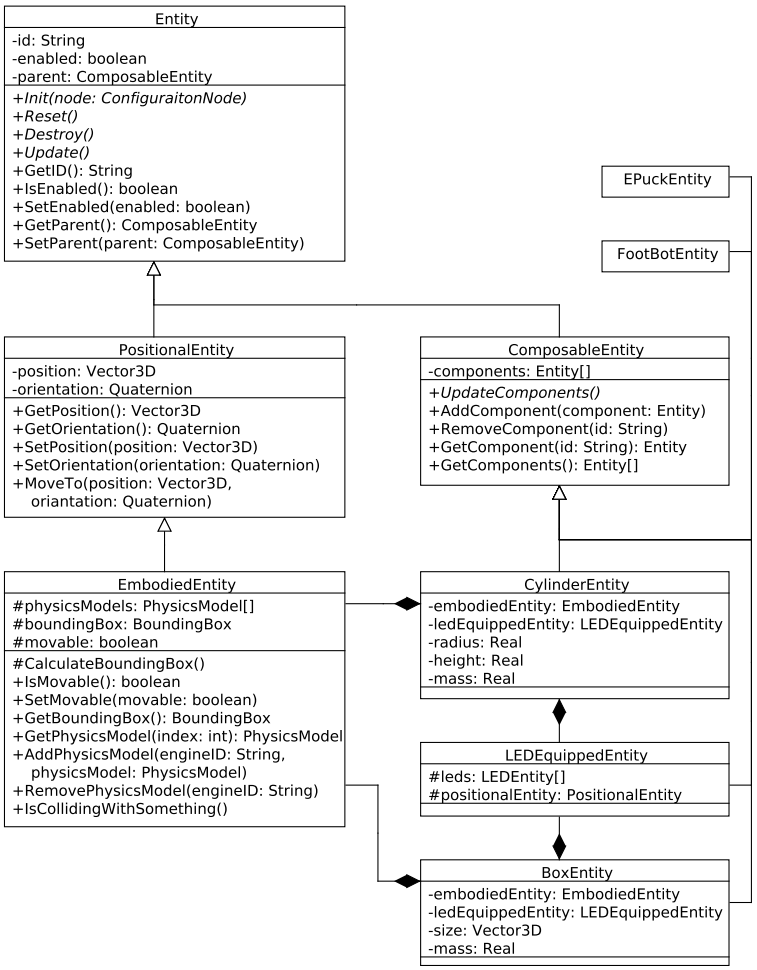

Figure 3.1: Example of ARGoS's entity hierarchy.

called entities, that contains the complete state of the simulation, i.e., the state of each object involved. An object can be a robot, an obstacle or any other item. To represent different kinds of objects, ARGoS offers several types of entities, where each of them stores a particular aspect of the simulation. Entities can be combined to each other, creating composable entities able to model more complex objects. Entities can be also created brand new extending the basic entity class. The state of a simple object can be stored in a simple entity like positional entity, that contains only the position and orientation of the object. For a wheeled robot, the sole positional entity is not enough. In fact, its state must be completed by an entity that stores the speed of the wheels, called in ARGoS wheeled entity, and a controllable entity that keeps a reference to the user defined control software and to the robot's sensor and actuators.

Entity types are organised in hierarchy. For example, Figure 3.1 shows that the embodied entity is an extension of the positional entity that adds a 3D bounding box for the referred object. Robots' entities are themselves extensions of composable entities since their state incorporates many aspects. Figure 3.2

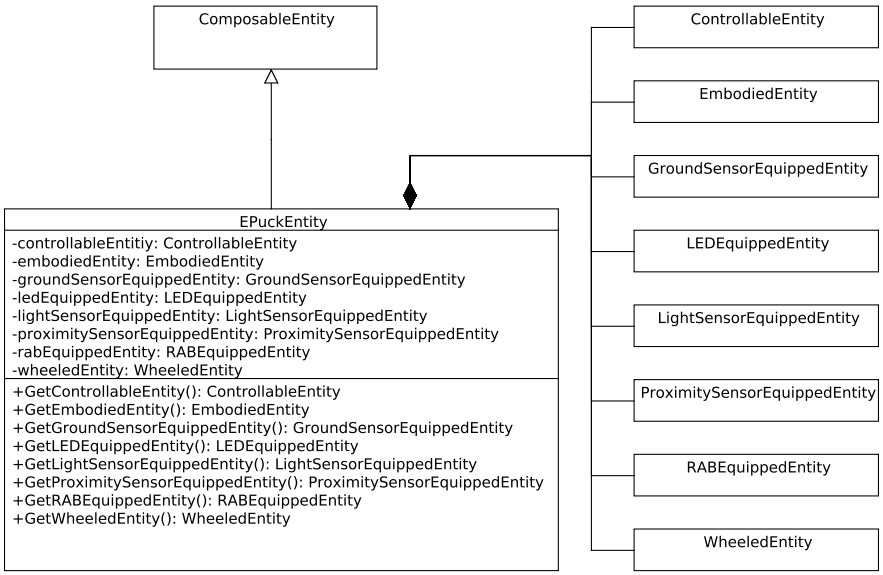

Figure 3.2: E-Puck entity class diagram.

depicts the types of the entities that compose the E-Puck entity. Such a design is essential to provide flexibility and at the same time promote code reusability and avoid redundancy.

Sensors and actuators are plugins that access the state of the simulated 3D space. Sensors access the space in read-only mode, while actuators are allowed to modify it. More precisely, every sensor and actuator is designed to only access the space entities involved in the specific measurement. The advantages of bounding sensors and actuators to specific entities affect both flexibility and efficiency. Flexibility is enhanced because tailoring sensors and actuators plugins to specific components instead of the whole robot often leads to a generic robot independent plugin design, that can be reused with different types of robots. Another benefit to flexibility is that robots can be built easily and quickly incorporating existing components, guaranteeing that all the sensors and actuators targeted for those components will work without modifications. In the matter of efficiency, the robot components associated to sensors or actuators that are not employed in a simulation do not need to be updated, preventing unnecessary waste of computational resources.

Physics engines are the components in charge of calculating the positions of the robots according to a set of consistent physical laws. In ARGoS, physics engines are plugins allowed to update the embodied entities status. ARGoS provides four kinds of physics engines: a 3D-dynamics engine based on Open Dynamics Engine (ODE), a 3D particle engine, a 2D-dynamics engine based on the open source physics engine library Chipmunk [1], and a 2D-kinematics engine. The researcher can choose the appropriate engine according to the kind of simulation he/she is carrying out. In addiction, ARGoS allows to split the 3D space assigning to each subspace a different physics engine. This feature permits a very fine grain of control of the 3D space. Furthermore, using several physics engines boosts the simulator performances because, thanks to ARGoS

multi-threading, the computation of different engines is parallelised.

Robot controllers are plugins that contain the control software of the robots. One of the most powerful features of ARGoS is the possibility to test a particular controller on real robots after the simulation without changing the control software code. This feature is very important because the researcher is able to compare the performances of the same controller in simulation and reality and can identify more precisely the source of possible discordances. ARGoS robot control software is written in C++, however, robots used for swarm robotics usually embed a low-end processor. Thus, before migrating the code from the simulator to the robots it is necessary to recompile the control software for the particular architecture. Apart from the recompilation, the control software does not need any refactoring. To guarantee the portability of the code, the controller designer relies on control interfaces to access the sensors and actuators. The control interfaces must be installed on the simulator side and on all the robot architectures, and must be implemented by all the sensors and actuators of the simulator module and the real robot modules.

Visualisations are plugins that read the state of the simulated 3D space and create the visual representation of it. At the moment three kinds of visualisations are available in ARGoS. Just like any other plugin component, the researcher can select the most appropriate visualisation according to the goal of the simulation. For high quality graphics ARGoS provides a rendering engine based on ray-tracing. For an interactive GUI the researcher can choose a plugin based on OpenGL. If the goal of the simulation is interaction with data analysis programs rather than visualisation, a text-based visualisation plugin is available to avoid useless graphical computation.

It is very hard to cover all the possible use cases for a general purpose simulator like ARGoS, without making the tool unmanageable under the usability and maintainability point of view. ARGoS design choice to grant flexibility, without burden the tool's complexity, is to provide user-defined function hooks in strategic points of the simulation loop. The user is able to write the so called loop functions, adding custom functionalities to be executed before or after the control step. A loop function can access and modify the whole simulation. In fact, the user is allowed to store data for later analysis, calculate statistics, and even create a new dynamic in the simulation by adding or remove objects in the 3D space.

Deep modularity and flexibility makes ARGoS suitable for execution on real robots, upon appropriate compilation. The components intended for the robots must be implemented accordingly and specifically compiled for the particular purpose. ARGoS code structure can be seen under two perspectives: deployment and source code. The deployment point of view groups the components according to their target, that can be simulator, robots, or both. This classification consists of three main areas: control interfaces, simulation modules and real robot modules. Figure 3.3 shows the structures and the contents of the different areas. In the control interface area is located the software needed both on simulator and on the robots, typically control interfaces for sensors and actuators. In the simulation area goes the whole set of components used by the simulator and by the simulated robots: simulated sensors and actuators, space entities, physics engines, and visualisations. The real robot area contains the code specifically implemented for the real robots. Typically the real robot area reflects the structure of the simulated robots part in the simulator area.

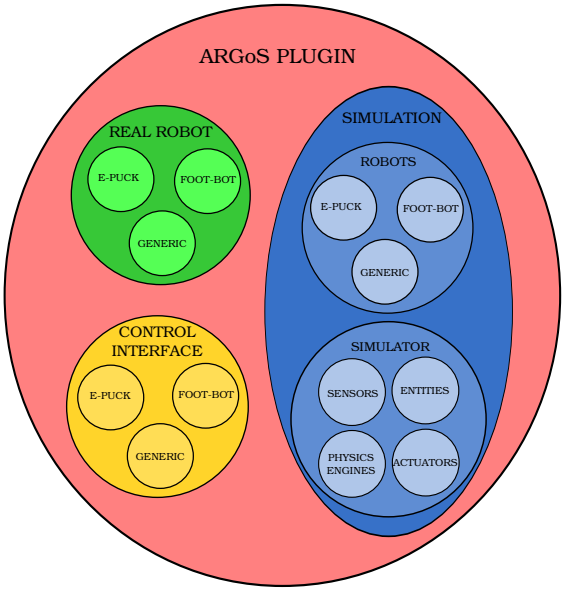

Figure 3.3: ARGoS deployment areas.

The components in this area are sensors and actuators and any other specific component that can execute on the real robots.

The source code perspective reflects the source code tree organisation. This perspective groups the components according to their logical function rather than the target. Figure 3.4 gives a representation of the source code tree, differentiating the distinct target areas with colours.

From the two classifications emerges that the robots section is divided into more subsections including generic, e-puck, and foot-bot. The generic section contains sensors and actuators tied to space entities that are so generic to be employed on any type of robot. To implement any robot specific sensor or actuator it is necessary to create the respective robot directory in the source code tree and implement the components for the desired target: common interface, simulator, or real robot. ARGoS currently provides the simulation and real robot modules for E-Puck robots and the simulation module for Footbot robots. To execute the code on real robots, the dedicate modules must be compiled for the specific robot architecture. In this case the built output comprehend the green and yellow areas of Figure 3.3, while when compiling for simulation target the resulting built output corresponds to the blue and yellow areas.

The ARGoS settings must be configured in an XML configuration file. The configuration file specifies the set up of the whole experiment, it contains information about the arena, the robots, the physics engines and the loop functions. The same XML configuration file is used for launching both the ARGoS simulator and the real robots main, therefore it must be uploaded on all the robots together with the control software.

The general structure of the file is given by a root node called argos-configuration and six child nodes: framework, controllers, arena, physics engine, visualisation, and loop functions. The framework node specifies some internal parameters used by ARGoS core, such as the number of threads used for multi threading,

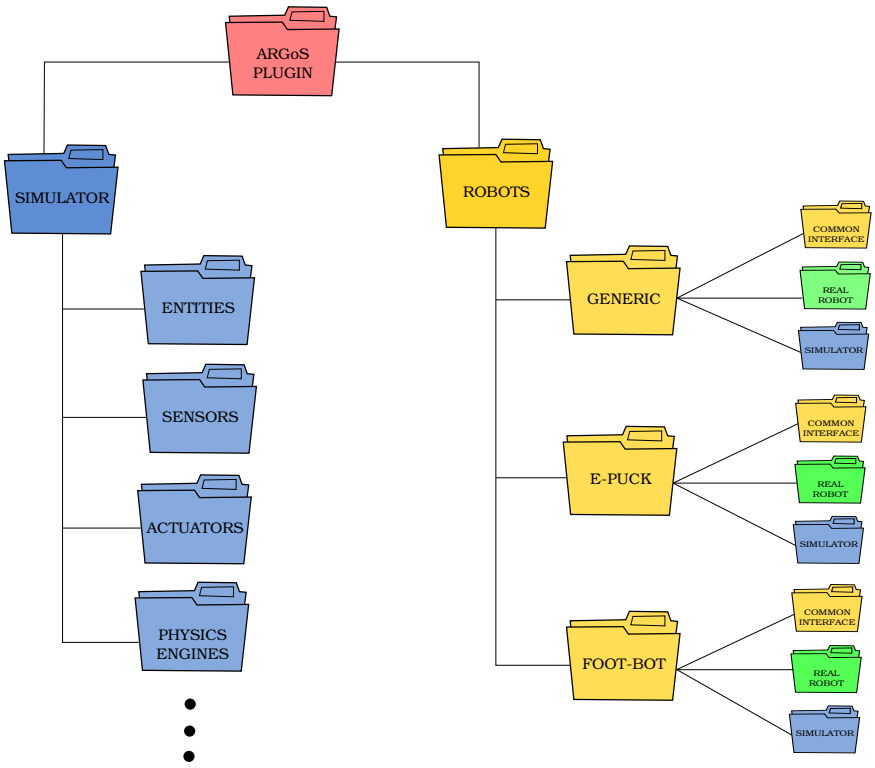

Figure 3.4: ARGoS source code tree. Target areas are grouped by colours: blue for simulation, green for real robots, and yellow for both.

the base random seed for the random number generator, the duration of the experiment, and the length of the control step. The controller node contains a list of user defined control software associated to a unique identifier. The arena node contains the entities to be placed in the arena at the beginning of the experiment. These entities include arena walls and robots. The robots are configurable with initial position and controller type. The physics engine node configure which physics engine to use. In this section the researcher can also declare more physics engines and their connections. The visualisation node selects which visualisation set up to use. Finally, the loop function node configures the user defined functions, that can execute before or after the control step for any application specific purpose.

The XML configuration file simplifies and speeds up the process of environmental set up. The configuration file can be quickly modified and it does not need recompilation to be effective. Efficiency also benefits from this configuration method, because ARGoS is able to load only the plugins defined in the configuration file, avoiding to load and execute useless components.

Given the general structure of ARGoS, let us now focus on the dynamics of an ARGoS execution session. When the simulator starts, it takes a few steps to load all the needed classes and create their instances in a precise order. The initialisation procedure parses the XML configuration file and instantiates the required entities. Each node initialisation triggers its components' initialisation

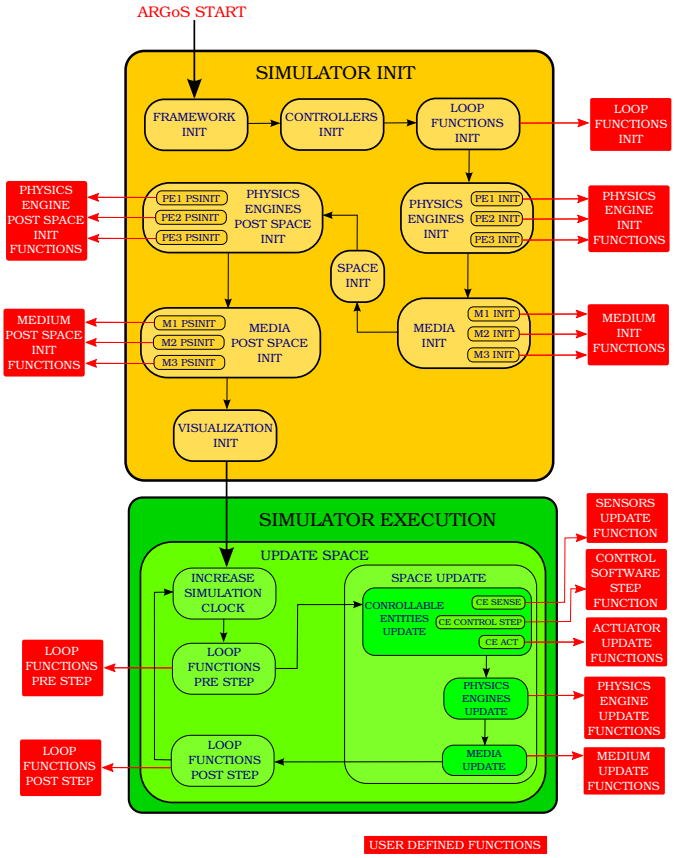

Figure 3.5: ARGoS main thread workflow. The black arrows represent the flow of the thread, the red boxes represent user definable functions. The red arrows are hooks for user defined functions executions.

in a chain. Figure 3.5 depicts the flow of the main ARGoS thread, explaining where, during the execution, the customisable user defined functions are hooked in.

The first node ARGoS initialisation explores is the framework node, then controllers, loop functions, physics engines, media, and finally space. The space entity must be created at last, however there are some entities that need to access the space to complete their initialisation. This is the case of physics engines an media. To this purpose, physics engines and media entities have the possibility to implement a second initialisation procedure which is able to access the space, called post space init. The visualisation is initialised at last, and after that ARGoS enters the execution loop.

The execution is demanded to the particular visualisation plugin in use. The visualisation plugin performs the execution loop for the duration of the experiment. Each loop of execution must include a callback to the simulator's *UpdateSpace* method. ARGoS accomplishes the space update through a well defined procedure, in which a chain of actions are triggered one by one on the components in a precise order. As first operation, the simulation clock

is increased. Then, any possible loop function that executes before the space update is called. After that, the space update takes place, followed by a call to all the loop functions to be executed after the control step.

The space update is a complex operation divided in turn in three main phases: controllable entities update, physics engines update and media entities update. Each update operation triggers the update of the entities belonging to the respective area. Controllable entities, the entities subjected to a control software, perform update in three steps: sense, control step, and act. Once again, the sensing procedure triggers the update function of each sensor employed, including virtual sensors. When all the sensors are updated, the control step is processed. The control step is the body of the user defined control software, it codes the behaviour of the controllable entities according to the updated sensor values and the behavioural logic implemented. After that, it is time for the actuators to operate, and that is what the third stage of the controllable entities update does.

Now the state of all the controllable entities is updated. Here starts the update of the physics engines and possible entities transfer between two physics engines. The physics engine update can be managed according to the type of engine needed. For example, the implemented 2D dynamics engine based on Chipmunk libraries first updates the state of the physics model from the entity state, then calculates the physical feasibility of the new physics model states altogether, and finally updates the space accordingly.

The same mechanism applies when ARGoS shuts down. In this case the components are deallocated in this order: loop functions, visualisation, space, media, and physics engines. The user defined function to decide how to deallocate a customised entity is the *destroy* function.

Given this quick and partial overview of some aspects of ARGoS, I am going to focus on virtual sensing, its motivations and benefits for the swarm robotics field, and how to realise such a technology using ARGoS as robot swarm simulator.

3.2 Virtual sensing with ARGoS

Virtual sensing technology in swarm robotics is the capability of real robots to perceive the features of a virtual environment. Among other things, such a technology aims to help researchers setting the environment for robot experiments. Currently, in many swarm robotics research labs, the experiment setting is a secondary although time consuming process, because often the arena space is handmade by the researcher. When the experiment involves a simple arena, the researcher can use the raw material available in the lab. However, even in this case the setting is not so trivial if some items must be placed in precise positions, or if objects must be attached together in a characteristic shape. The difficulty increases when very specific objects must be built or bought. In this case the cost and time spent for building the arena can be unaffordable, preventing to run real robots experiments in that wanted scenario.

A tool that provides virtual sensing comes in handy to speed up the arena setting. With virtual sensing, the researcher can simulate all or part of the environment, and make it real for the robots in a sort of augmented reality. Furthermore, the simulated environment can be physically or realistically infea-

sible. For example, it is possible to place a light source 10 m above the arena in a room where the ceiling is 3 m tall. In addiction, the virtual light source does not need any support nor installation to stand 10 m above the ground, unlike any other physical object. An even more appealing possibility is to create an environment and make it change in time by adding, removing or modifying objects. For example, if in an experimental set up the arena forms a labyrinth, it is possible to open and close paths during the robot exploration to test the adaptability of the swarm to a changing scenario. Another example of what can be done with augmented reality, is to test robots aggregation on coloured spots that change position or phototaxis with a light source that moves around the arena. Virtual sensing makes this kinds of experiments feasible without need of mechanical infrastructures.

Another unfortunate but common case in swarm robotics labs concerns the lack of reliability of the robots and their sensors. Usually the labs dispose of a redundant set of robots for swarm experiments. However, due to the minimalistic architecture of the robots employed in swarm robotics, hardware failures are quite common and sensors reliability is sometimes questionable. Detecting sensor failures may be challenging, especially if the sensor keeps working exhibiting a random behaviour. In case a large part of the swarm carries faulty sensors, the experiment results can be heavily adulterated. Dealing with hardware sensors is laborious even with fully efficient sensors, because they must be periodically calibrated. Calibration is an operation that transforms the values perceived by the hardware sensor into meaningful numbers for the control software. For example, different ground sensors can generate different values for the same ground colour. Calibration tunes the scale of the different sensors in order to give the same value when perceiving the same ground colour. Sensors calibration is an operation that the researcher must perform on each robot, and it might be quite time expensive if the swarm is oversized. Virtualising sensors is a solution that allows researcher to quickly set up and start the experiment, avoiding the annoying and time wasting sensors calibration phase and possible sensor faults before or during the execution of the experiment.

Virtual sensing technology is not only suitable for replacing faulty sensors. In fact, a very interesting application of this tool is to create sensors that are not mounted at all on the available swarm. This application of the virtual sensing technology is useful for prototyping new sensors that might be installed on the robots in the future. Sensor prototyping can be applied to unavailable sensors or existing sensors that the researcher wants to enhance to achieve a certain level of reliability. A typical scenario for sensor prototyping is when the swarm needs to be upgraded with new or more performing sensors. Before buying or building the new hardware, implementing the virtual version of the sensor in object helps the researcher to tune the sensor's parameters and identify the best trade off between cost and performance for the targeted level of efficiency. The researcher is able to test the same sensor at several degrees of quality, simply changing the virtual sensor's parameter. It is a fast and cheap way to identify the less expensive sensor that permits a predetermined level of swarm's performance.

Sensor prototyping is particularly useful in swarm robotics because of the high number of robots to be upgraded. The costs and time to install the new hardware on the whole swarm can be critical, and it is very important that the new sensor works as expected to avoid waste of time and money.

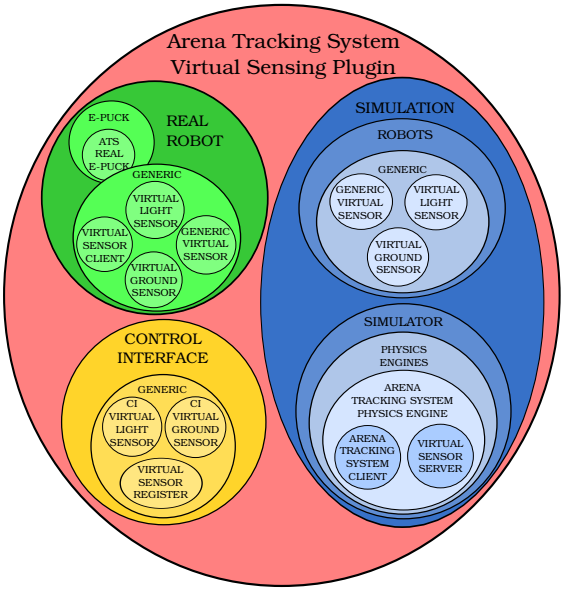

Figure 3.6: Arena Tracking System Virtual Sensing Plugin components divided in target areas: blue for the simulator module, green for the real robot module, and yellow for the control interface shared by both modules.

Virtual sensing technology basically creates a virtual environment and allows real robots to perceive it. The virtual environment is built within a simulator, in this case ARGoS. The choice of ARGoS is justified by its extreme flexibility. The plugins structure of ARGoS makes extremely easy to add new components with the only requirement to implement some specific interfaces.

Our realisation of virtual sensor technology employs a physics engine plugin, the Arena Tracking System Virtual Sensing Plugin, ot ATS-VSP (see Figure 3.6). The ATS-VSP is a container for several components, including the Arena Tracking System Physics Engine, or ATS-PE, that provides the real robot positions in the arena to the ARGoS core. The robot positions come from the ATS-S, which ARGoS accesses thanks to a corresponding Arena Tracking System Client (ATS-C). The simulator then executes the control step and calculates all the sensor values, including virtual sensors. Virtual sensor simulator modules are particular types of sensors implemented ad-hoc to store their values in a data structure called Virtual Sensor Data Structure or VS-DS. At the end of the sensors update process, when the VS-DS has been filled, the ATS-PE takes responsibility to dispatch the virtual sensors data to the corresponding real robot through Wi-Fi connection. When the real robots receive the virtual sensor data, the virtual sensor real robot modules installed on the robots pick their corresponding data and set it as sensor reading value. The robot control software accesses the virtual sensor's value through the specific virtual sensor control interface implemented by the virtual sensor that provides a method to get the sensor's reading. The virtual sensor control interface mechanism behaves the same way as a regular sensor control interface, achieving transparency form the controllers' point of view.

The communication infrastructure between the ATS and ARGoS, and be-

tween ARGoS and the robots is partially embedded in the ATS-PE and partially in the real robot module of the ATS-VSP. In the next two sections I describe the architecture of the ATS-VSP and its contents, the physics engine, the virtual sensors, and the components that form the communication infrastructure from the simulator and real robots perspective.

3.3 Arena Tracking System Virtual Sensing Plugin Simulator Module

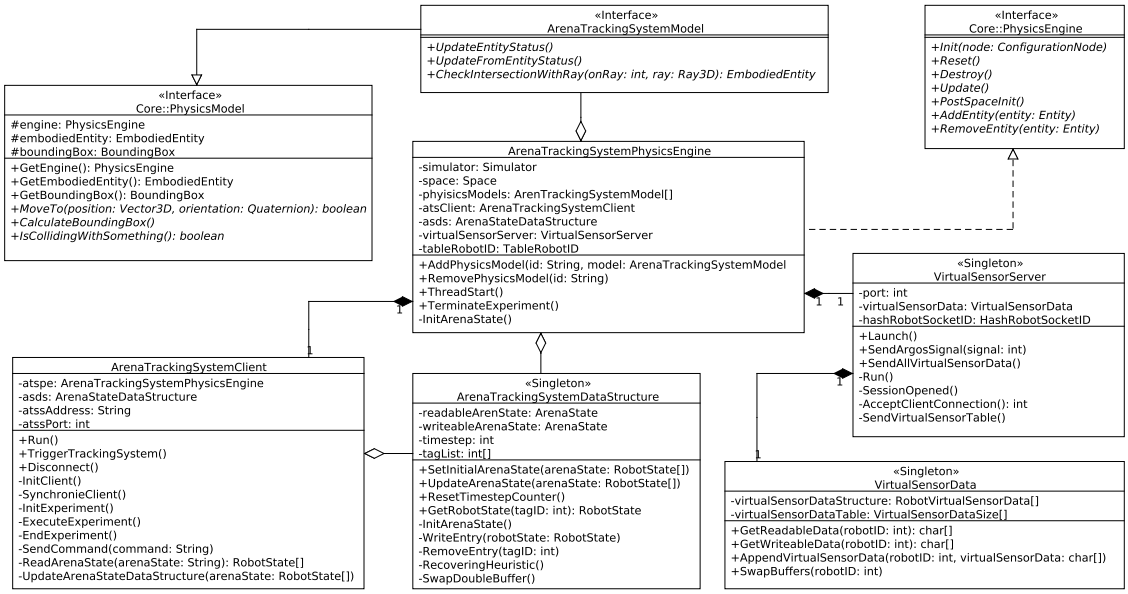

Figure 3.7: ATS-VSP Simulation Module class diagram.

The virtual sensing tool created for ARGoS is enclosed in a plugin called Arena Tracking System Virtual Sensing Plugin, referred from now on as ATS-VSP. The plugin includes the physics engine, already known as ATS-PE, the generic virtual sensor interface, and the components necessary to establish the communication between the three macro entities: Arena Tracking System, ARGoS and the robot swarm (see Figure 3.7). In addiction, two types of virtual sensors have been implemented and included in the ATS-VSP: the ground virtual sensor and the light virtual sensor. Chapter 4 exemplifies the creation of new virtual sensors and can be used as a guide for building new virtual sensor plugins.

This section describes the simulator module of the ATS-VSP, that is the part compiled and execute on ARGoS side (the blue and yellow areas in Figure 3.6). The next section discusses the real robots module, i.e. the result of the compilation for a specific robot architecture (the yellow and green areas in Figure 3.6).

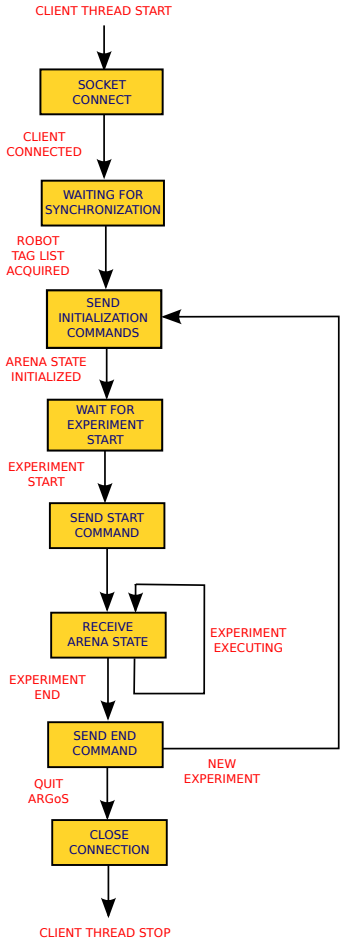

Figure 3.8: ATS-C workflow.

3.3.1 Arena Tracking System Client

The Arena Tracking System Client, namely ATS-C, is the component that establishes the connection with the ATS, through the Arena Tracking System Server. The operations of the ATS-C are executed on a thread forked from the main ARGoS thread during the initialisation of the ATS-PE. The creation of a new thread is essential to handle the communication and the data exchange in real time with ATS-S, while ARGoS is executing its own procedures. The life cycle of the ATS-C thread is exemplified in Figure 3.8.

As soon as the ATS-C starts, it connects via TCP socket to the network host and port specified in an attribute of the framework node in the XML configuration file. The researcher must therefore know the IP address of the machine hosting the ATS-S, and which port the server bound. At the moment of the execution of the ATS-C, the ATS-S must be already running and ready to accept a client connection. Right after connection there is a phase of synchronisation between the ATS-C thread and the ARGoS main thread. This synchronisation is due to the necessity of initialise part of a shared data structure, called Arena

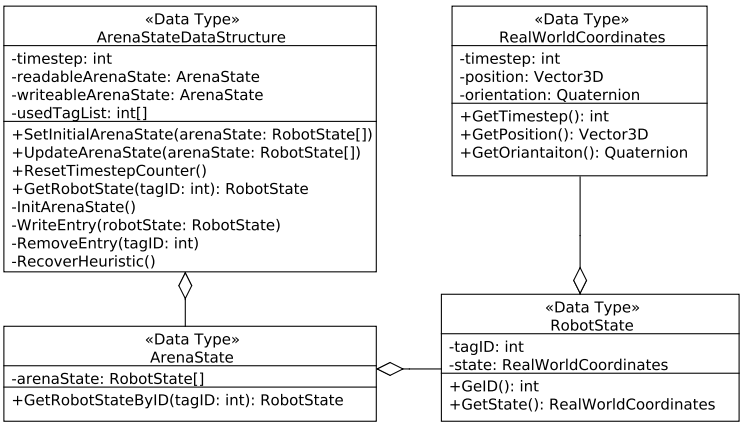

Figure 3.9: Arena State Data Structure diagram.

State Data Structure, or AS-DS. The AS-DS is used to store and share the real time updated arena state between the ATS-C and the ATS-PE, and among other data it contains a list of the robot tags employed in the experiment (see Figure 3.9). To improve reliability against ATS possible detection or decoding errors, the ATS-C executes a heuristic algorithm to fix ATS failures that uses the tag list. This list is inserted in the AS-DS by the ATS-PE, inquiring the list of entities available in the space that are defined in the arena node of the XML file. This operation cannot be done before the space is initialised, therefore the tag list is available for the ATS-PE only in the *PostSpaceInit* phase. In other words, the experiment cannot be initialised before the ATS-PE obtains the tag list.

Achieved the synchronisation with the ATS-PE thread, the ATS-C switches to an experiment initialisation state in which it sends commands to the server to get the starting state of the arena. The scenario is as follows: the ATS-S is running, the ATS-C is connected and the robots are placed still on the arena. ARGoS is ready to give the representation of this static picture. The ATS-C sends the command *one_shot* (see section 2.3.2) to receive the arena state only once without incrementing the timestep counter. Once the initial arena state is obtained and displayed in the ARGoS GUI, the ATS-C enters a waiting state. When the researcher decides to start the experiment giving the play command on the ARGoS GUI, it automatically causes the ATS-C to exit the waiting state and trigger the ATS-S experiment execution state with the command *experiment_start*.

From this moment to the end of the experiment, the ATS-C enters a loop in which repeatedly waits for the new arena state, deserilises the received data and updates the AS-DS. The experiment terminates in two possible ways: the time set as experiment duration expires, or the researcher interrupts the experiment with the button stop on the ARGoS GUI. In both cases, ATS-C exits the loop and sends the *experiment_stop* command to the ATS-S, which in turn ends the tracking session. Now the ATS-C enters a new state waiting the researcher's input. If the researcher sets another run with the button reset in the ARGoS GUI, the ATS-C returns in the experiment initialisation state. In case the researcher quits ARGoS, the ATS-C disconnect from the ATS-S and terminates.

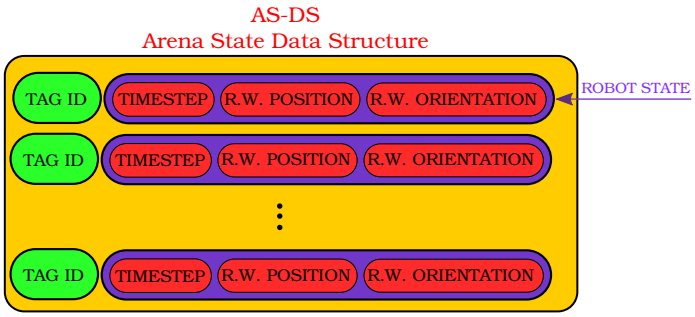

Figure 3.10: Arena State Data Structure.

The AD-DS is the shared data structure designed as a singleton that stores the arena state. The arena state is defined as a list of pairs, where the first element is an integer representing the tag ID of the detected robot, and the second element is the corresponding robot state data structure (see Figure 3.10). The robot state data structure is composed by real world position and orientation of the robot, and its referring timestep.

As already mentioned, the AS-DS is shared between two processes: ATS-C in write mode and ATS-PE in read only mode. The main issue with the shared data structure is that the ATS-PE reads it every 100 ms, while the ATS-C writes on it as soon the ATS-S sends a new arena state. For the reasons explained in section 2.2.1, the ATS-S arena state cycle time is neither constant nor predictable. To grant data consistency, the threads must access the shared data structure in a mutually exclusive way. The AS-DS provides a mutex system do avoid race conditions. However, the soft real time constraints of the ATS-PE require a mechanism to minimise possible blocks for accessing the shared AS-DS. In other words, it is undesirable that the ATS-PE reads the AD-DS while the ATS-C is already writing on it. To prevent ATS-PE blocking, the writing time of the ATS-C on the AS-DS is minimised thanks to a double buffer data structure design. The AS-DS actually stores two versions of the arena state, a readable stable version that is accessible only by the ATS-PE, and another that can be accessed only by the ATS-C in write mode. The two concurrent entities can access the buffers only through pointers, one for the readable buffer, and another for the writeable buffer. When the writeable buffer is ready, i.e. the ATS-C finished to write the updates, the pointers to the two buffers are swapped. The pointer to the readable buffer, hold by the ATS-PE, refers now the newly updated buffer, and the pointer to the writeable buffer points to the buffer that is now outdated and can be rewritten. The swapping operation must be executed in exclusion with the ATS-PE reading in any case, but the execution time of switching is not comparable with the execution time of the AS-DS update operation. Figure 3.11 shows five possible scenarios in which the two threads try to access the AS-DS. Scenarios A and B: the ATS-PE is the only thread accessing the AS-DS and it retrieves the updated version of it. In scenario C, the ATS-PE access the AS-DS in the yellow zone, that is when ATS-C is writing but not locking the resource. ATS-PE can lock and read the data from the ready buffer, which contains the previous version of the arena state. ATS-C will be able to lock the AS-DS for pointer swap only after the

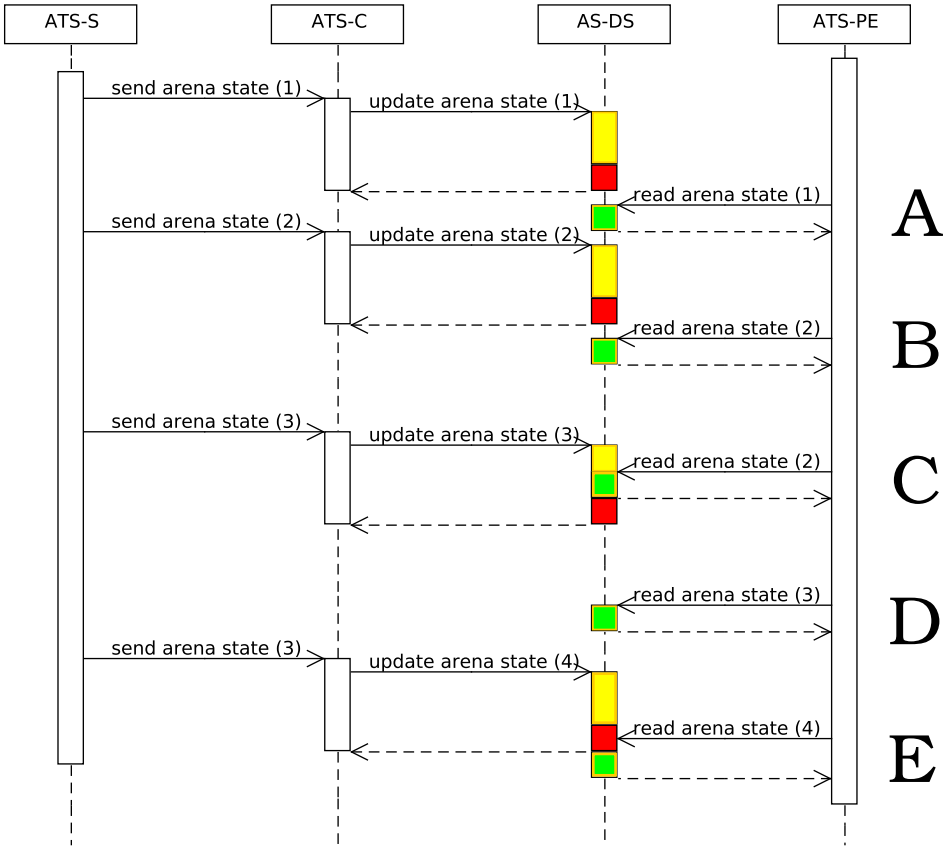

Figure 3.11: Sequence diagram focused on the shared resource AS-DS. Red box indicates that the resource is locked by the ATS-C thread, green box indicates that the resources is locked by the ATS-PE thread, and yellow box indicates that ATS-C is updating the double buffer without locking the resource.

ATS-PE has released it. In this case, ATS-PE reads twice the same arena state from AS-DS (scenarios B and C). In scenario D, ATS-PE is the only thread accessing the AS-DS and it reads for the first time the updated arena state version. Scenario E shows the unlikely event in which the ATS-PE needs to access the AS-DS when it has already been locked by the other thread ATS-C. This is a scenario that blocks the execution of the ATS-PE, although only for the time needed to swap the pointers to the double buffer. As soon the ATS-C has released the resource, the ATS-PE can lock and read it.

Without the double buffer mechanism, the ATS-C must lock the AS-DS also in the yellow zone. In scenario C, for example, this would lead to a significant delay in the arena state update in ARGoS, creating an unacceptable gap between the real dynamics of the experiment and the simulation. Eventually, even in the unlikely scenario E, the time ATS-PE wastes in the block is negligible.

3.3.2 Arena Tracking System Physics Engine

The Arena Tracking System Physics Engine, or ATS-PE, is the entry point for the virtual sensing tool in ARGoS. As explained in Section 3.1 and depicted in Figure 3.5, ARGoS provides hooks for execution of user defined functions at several stages. The ATS-VSP is a physics engine plugin. In other words, it extends the ARGoS core class *PhysicsEngine* and implements a physics model class, that must be extended by each physics entity managed by the physics engine. By implementing the required interfaces, the ATS-VSP acquires the physics engine plugin status. This means that if the ARGoS XML configuration file defines a physics engine of type Arena Tracking System Physics Engine, an instance of the ATS-PE is created and the user defined functions are executed all in good time.

The first execution of the instance of the ATS-PE happens when ARGoS calls all the physics engines *init* operation. In the initialisation phase, the ATS-PE parses the *framework* node of the XML configuration file and gathers necessary information to instantiate the networking support components: ATS-C, already discussed in Section 3.3.1, and the Virtual Sensor Server, or VS-S, described in Section 3.3.3. The XML file must include the address and port of the host where the ATS is running, and the port on which the VS-S must execute on the ARGoS host. This information must be written in the XML file by the researcher, in form of attributes of the *framework* node. The attribute names are: *ats_host*, *ats_port*, and *vss_port*. At the end of initialisation, the ATS-PE creates the instance of the ATS-C and spawns a new thread on which executes a method of ATS-C that plays the role of client's main. Since now on, the ATS-C executes on its own thread, leaving the ARGoS main work flow unaffected.

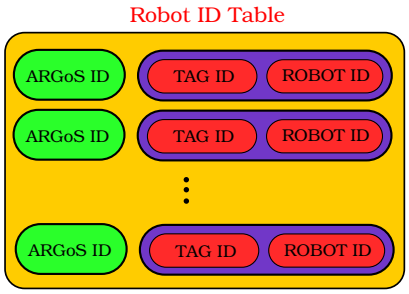

Figure 3.12: The ID table data structure.

While the ATS-C is running on a separate thread, the ARGoS main flow continues, and the ATS-PE executes again during the ARGoS physics engines post space init phase. The ATS-PE exploits the *PostSpaceInit* function to bind the tag ID of the robots to a pair composed of the robot ID, i.e. the last part of the robot's IP address, and the argos ID, i.e. the name used by ARGoS to identify a physics entity. The table containing the triple made of ARGoS ID, tag ID and corresponding robot ID is necessary to univocally identify a single robot under every point of view (see Figure 3.12). In the execution process the ATS-S provides to the ATS-C a list of tag IDs. The ATS-PE must be able to identify the physics entity associated to the robot with a particular the tag ID. Finally, the VS-S must send the virtual sensor data to the network IP address

corresponding to the robot with that particular tag ID on top. This triple association is essential and must be cured by the researcher. The cornerstone of the triple association resides in the robot ID attribute of the configuration file. In order to guarantee the consistency of the execution, the researcher is forced to write the robot ID attribute in this precise form: *argosID_tagID_robotID*. The ARGoS ID can be an arbitrary string, while the tag ID must represent the decimal value of the tag, and the robot ID is once again the last part of the robot IP address. When the triple table is completed, ATS-PE performs the update of the arena state and the space to produce the correct visualisation of the initial arena state. To complete the post space initialisation phase, the VS-S is launched and another thread is spawned to host it. The VS-S is the component that manages the communication between the simulator and the robots. Since its creation, the VS-S waits indefinitely for robot connections. When a connection is detected on the listening port, VS-S accept it, sends some initialisation data to the robot and stores the socket descriptor to recover the communication channel when needed (see Section 3.4.1).

Before starting the experiment, all the robots must be connected to the VS-S. To obtain this result, before commanding the beginning of the experiment the researcher must execute the ATS Real E-Puck main on all the robots involved in the experiment. The ATS Real E-Puck main is a version of the Real E-Puck main modified to establish a connection with the ARGoS host. The connection to the VS-S is performed by a client, called Virtual Sensor Client, or VS-C, running on each robot, once again on a dedicated thread (see Section 3.4).

The next ATS-PE function triggered by ARGoS flow is the update function. The update function is executed periodically until the end of the experiment. The command that starts the experiment is given by the researcher clicking the play button on ARGoS GUI. This event triggers a set of operations throughout the system. As stated above, the ATS-C exits its waiting state and ask the ATS-S to start acquiring and processing images. On the other side, the VS-S sends to all the VS-C connected the instruction to start the robot execution, kept on hold by the ATS Real E-Puck main. The result is that at play signal, all the components of the system automatically enter the execution state. The robots start to move according to their control software, the tracking system begins to produce arena states for ARGoS, and all the components in ARGoS execute the periodic update function.

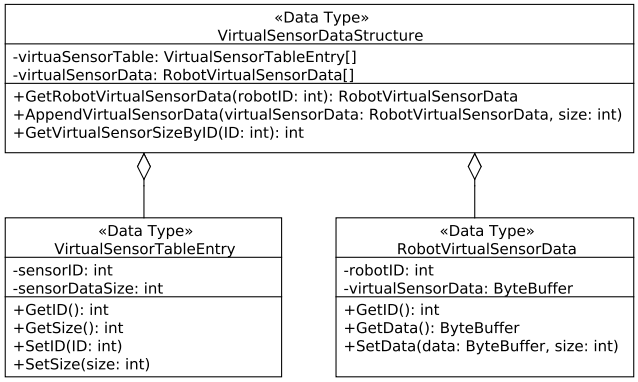

Figure 3.13: Virtual Sensor Data Structure diagram.

The body of the ATS-PE *Update* function is quite simple, in fact it basically consists of cycling the physics models belonging to the engine, call the *UpdateEntityStatus* function and update the virtual sensor data for the specific robot. The updating of the physics entity status implies the update of all the sensors belonging to the entity, including virtual sensors. When a virtual sensor is updated, it writes the reading in the Virtual Sensor Data Structure, called from now on VS-DS. The VS-DS is a singleton designed list of buffer containing the virtual sensor data for each robot, indexed by the robot ID. When all the virtual sensors update their value on the VS-DS the ATS-PE asks the VS-S to send each Robot Virtual Sensor Data entry to the corresponding robot (see Figure 3.13).

The experiment execution terminates either when the beforehand set time expires, or when the researcher clicks on the stop button on the ARGoS GUI. When this happens, the ATS-C sends the stop experiment command to the ATS-S to quit the tracking session and the VS-S sends a termination code to the robots. The researcher now can reset the experiment with the reset button on the ARGoS GUI or quit the application. In both cases ARGoS triggers the respective function in a chain on all its components. In the first case the function is called *Reset*, in the latter is called *Destroy*. The simulator reset function aims to restore the state of the experiment after the post space initialisation phase. As a consequence the reset function of the ATS-PE re initialises the AS-DS and sets all the additional data to be ready for a new experiment. The destroy method instead deallocates all the previously allocated data structures and disconnects the ATS-C form the ATS-C.

The update function of the ATS-PE is the cornerstone of the virtual sensing technology, the procedure that ties together the ARGoS's upstream and the downstream. Its simplicity derives directly from the extremely flexible architecture of ARGoS, that defines a solid execution framework in which it is possible to add custom components and functions.

3.3.3 ARGoS Virtual Sensor Server

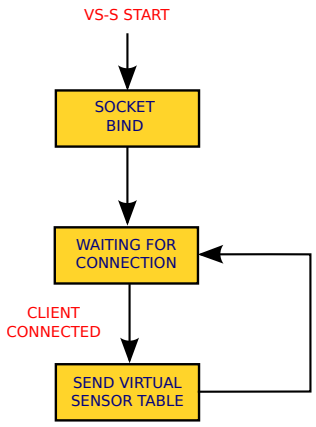

Figure 3.14: Virtual Sensor Server fork flow.

The Virtual Sensor Server, or VS-S, is the component that handle the com-

munication between ARGoS and the robots. The VS-S runs on a dedicated thread spawned by the ATS-PE in the post space initialisation phase. Its simple work flow (see Figure 3.14) has the sole purpose of receiving robot connections, store the socket descriptor in a data structure called Robot-Socket Hash Table, and sending back a preliminary piece of data called Virtual Sensor Table, or VST. The Robot-Socket Hash Table, or RS-HT, is a hash table data structure shared between VS-S and ATS-PE that contains the socket descriptor of a robot connection channel indexed with the robot ID (see Figure 3.15). The access to the hash table is made thread safe by using a mutex. The VST is a hash table

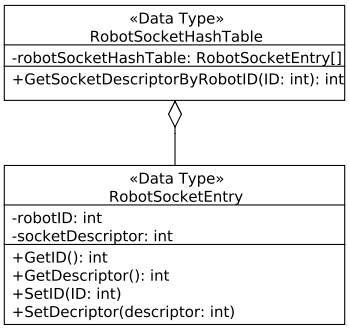

Figure 3.15: Robot-Socket Hash Table diagram.

part of the bigger VS-DS (see Figure 3.13). The VST contains a sensor ID and the size of its reading in bytes. The sensor ID is statically defined in another data structure called Virtual Sensor Register, while the reading data size computation is demanded to the specific virtual sensor in its initialisation. The VST is filled by the virtual sensors during the space initialisation. The hash table is sent to the robots right after the connection establishing. The VST is used by the VS-C to create a robot specific virtual sensor data structure. The Virtual Sensor Register, or VSR, is another hash table data structure where one element is the virtual sensor name as it must be written in the configuration XML file, and the other is the sensor ID, a statically assigned integer value (see Figure 3.16). The VSR utility is to automatically provide the same sensor ID to the corresponding virtual sensors both in simulation and real robot modules, hard coding the sensor ID only in one component. To do this the VSR needs to be part of the common interface and being compiled for both the simulator module and the real robot module.

The VS-S class implements two methods that are called by the ATS-PE on the main ARGoS thread. The purpose of the both of them is to communicate some information to the robots. One method is called *SendArgosSignal* and it is used to send the start, stop and reset commands to the robots when the researcher clicks on the ARGoS GUI buttons. The other method is called *SendAllVirtualSensorData* and it is the one that truly embodies the virtual sensing. This method sends the Robot Virtual Sensor Data, or R-VSD, to the respective robots. The VS-DS contains the most updated virtual sensor data divided for robot. Each entry of the VS-DS contains the robot ID and the R-VSD, a byte buffer that in turn contains the data of all the virtual sensors. During the ATS-PE update function, the virtual sensors append their updated reading in byte array format to the R-VSD byte buffer, using the first byte to specify

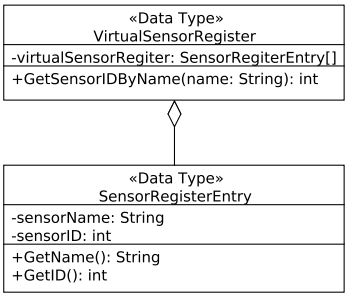

Figure 3.16: Virtual Sensor Register diagram.

their virtual sensor ID. At the end of the ATS-PE update function, the VS-S is asked to deliver the R-VSD byte buffer to the corresponding robot, retrieving the socket descriptor from the RS-HT.

Notice that the VS-S is simultaneously running the server loop on a separate thread, thus the RS-HT may be modified dynamically if some robot connect or disconnect from the VS-S. This feature increases the system flexibility and makes it suitable for experiments in which the robots are added or removed dynamically to change the swarm size at runtime.

3.3.4 Virtual Sensors Simulator Module

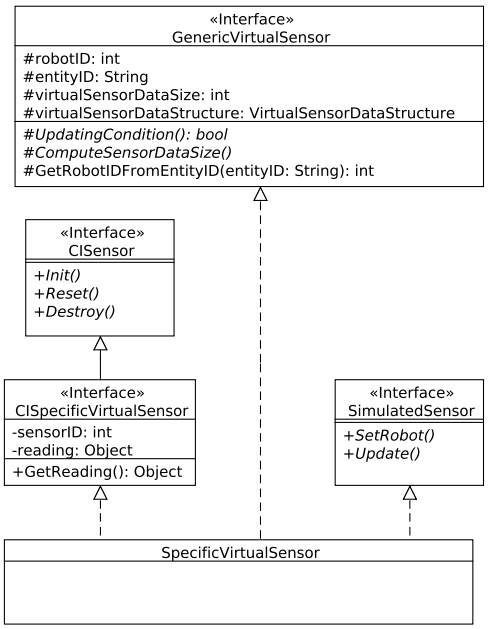

Figure 3.17: Virtual sensor class diagram.

The virtual sensors implemented for the simulation module are the components in charge of calculating the sensor reading exploiting the simulator information. The reading value is then serialised and appended in form of byte

array to the VS-DS, being the first byte the ID of the virtual sensor taken from the VSR. The reading value serialisation allows the virtual sensor designer to envision a sensor data structure of arbitrary complexity. The deserialisation of this data is performed by the real robot module counterpart of the virtual sensor, thus it only needs to reverse the serialisation made earlier.

Once again, the ARGoS protocol allows the sensors to execute user defined functions. Every virtual sensor must implement three interfaces to be considered a virtual sensor by ARGoS (see Figure 3.17). One interface to be implemented is the specific sensor's control interface, that the virtual sensor designer must provide as well. This interface gives the *sensor* status to the component. The specific virtual sensor control interface extends the more generic sensor control interface and it is implemented by both the simulator module and the real robot module. The specific virtual sensor control interface provides an abstract method to access the sensor's readings that must be implemented in the virtual sensor class. The control interface is exploited for the sake of portability of the control software. In fact, by using the sensor control interfaces it is possible to reuse the control software code from the simulator to the robot without refactoring, because the accessor method to the sensor's reading is defined by the control interface, but implemented by the proper underlying module.

The second interface to implement is the simulated sensor interface. Implementing this interface means that the sensor is intended to work for the simulator module, in opposition with the real robot module. The simulated sensor interface defines two abstract methods to implement: *SetRobot* and *Update*. The method *SetRobot* is used to bind the sensor's entities to the robot's physics entities. If the desired virtual sensor already exists as simulated sensor, it is possible to extend this particular simulated sensor class instead of the generic simulated sensor, and reuse the method implemented in the parent class.

The *Update* method is where the new sensor reading is computed, serialised and appended to the VS-DS. The virtual sensor designer implements the sensor reading procedure exploiting the entities that the virtual sensor is entitled to use and which have been associated to the robot's entities in the *SetRobot* method.

The third interface to implement is the generic interface for the virtual sensors called *GenericVirtualSensor*. This interface gives the status of *virtual* to the new sensors. Every virtual sensor owes by default a reference to the single instance of the VS-DS, and a method to calculate the robot ID and tag ID of the robot they belong to. In addiction, the virtual sensor interface forces each virtual sensor to implement two methods, *ComputeSensorDataSize* and *UpdatingCondition*. The first method must be called in the virtual sensor initialisation method in order to correctly set the sensor entry in the VST. The computation of the reading data structure size is left to the virtual sensor designer. The method *UpdatingCondition* defines the condition needed for the new sensor reading to be appended to the VS-DS and then sent to the robot over Wi-Fi network. The purpose of this condition is to unburden the data load of the VS-DS to be sent. For each robot, the amount of data regarding the virtual sensing sent every timestep depends on the number of virtual sensors defined and the size of their data. In the worst case, the R-VSD intended for a robot can cross the maximum transmission unit limit (MTU) for the wireless network. This unfortunate case forces the underlying protocol to send the data in two separate chunks, introducing a unwanted delay in the system. Because the operation is repeated for each robot, the system delay at each timestep can

be unbearably significant for the system's soft real time constraints. In this case, but also in general, it is desirable to deliver only the meaningful data to prevent network congestion. With the method *UpdatingCondition* the virtual sensor designer is able to subordinate the deliver of the virtual sensor data to the condition specified in the method. The condition can concern the reading value, or the update rate of the virtualised sensor, according to the behaviour implemented in the robot's control software. For example, the virtual sensor designer can choose that a specific sensor data is meaningless if closer than a given threshold percentage to the last reading sent. In this way the system saves bandwidth, and the the real robot keeps reading the last received data, which is close enough to the real data for the controller's purpose. Another use for the updating condition is to decrease the update rate of certain sensors. If the simulated environment conditions of the experiment do not demand a fine sampling of the given sensor's measure, the updating condition can be verified only once in a number of timesteps. For example, if the virtual environment includes only a fixed static lamp as light source, the light virtual sensor can update every 1 s rather than 100 ms, i.e. once every 10 timesteps.

When the component implements the three interfaces it becomes a virtual sensor for simulator module. To complete the utility it is necessary to implement its counterpart for real robots, since in ARGoS every virtual sensor consists of a simulator module and a real robot module. The next section describes the components belonging to the real robot module. In particular, the focus is on the E-Puck robot implementation, because the tool is targeted on that specific robot architecture.

3.4 Arena Tracking System Virtual Sensing Plugin E-Puck Module

In this section, I describe the structure of the ATS-VSP intended for real robots, in particular for E-Pucks. Recalling Figure 3.6, this section focuses on the green areas, while the blue areas have been presented in the previous section. As stated above, the yellow areas include the common software between the two modules, and it has also been commented in Section 3.3.

The ATS-VSP real robot module includes both generic elements and E-Puck specific elements. The generic elements are usually sensors and actuators that avoid any robot specific feature. This kind of components should be compatible with any robot architecture, sparing to enter into platform specific details. On the contrary, the architecture specific elements are usually sensors and actuators that feature specific characteristics of the hardware mounted on the robot. These kind of components cannot be used on robot architectures other than the ones they are built for.

The E-Puck specific components of the ATS-VSP real robot module are reduced to the bone. In fact, the idea is to let virtual sensing technology open to any robot architecture, implementing only the least possible robot specific elements. Therefore, the virtual sensing technology's bare bones reside in the generic package. There is situated the Virtual Sensor Client, the generic virtual sensor real robot module interface, and the virtual sensor real robot modules. For the specific robot architecture, the virtual sensor designer is required to

realise a proper main program and the robot specific classes to handle all the real sensors and actuators of the robot.

A full-featured E-Puck plugin for ARGoS has been developer at IRIDIA, and the ATS-VSP extensively draws from it. In the ATS-VSP plugin, the software for handling the real E-Puck in the ATS environment is reduced to a single component that extends the real E-Puck class of the E-Puck plugin with the necessary ATS features. The ATS version of the real E-Puck entity is called Arena Tracking System Real E-Puck, or ATS-RE. Thanks to inheritance, the ATS-RE can use all the real sensor and actuator handlers implemented in the E-Puck plugin, remaining the only E-Puck specific component in the ATS-VSP.

In the remainder of this section, I am going to discuss the networking component hosted on the robot architecture, namely the Virtual Sensor Client, the virtual sensors' real robot module, and the component that binds the networking to the virtual sensing, i.e. the Arena Tracking System Real E-Puck.

3.4.1 Virtual Sensor Client

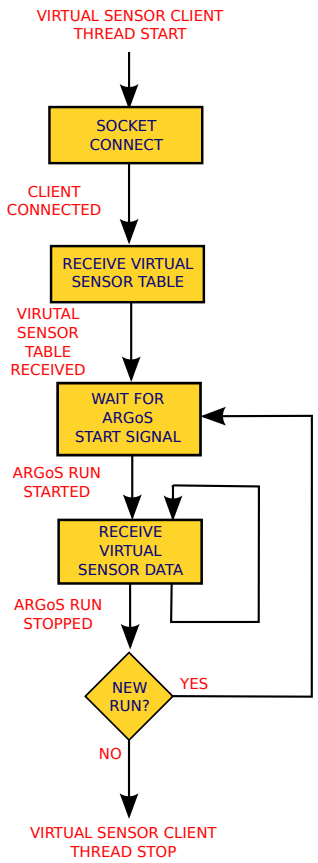

Figure 3.18: Virtual Sensor Client work flow.

The Virtual Sensor Client, or VS-C, is the networking component that must be installed on each robot of the swarm to complete the virtual sensing data

flow. Each robot must create a single instance of the VS-C, and make it run on a dedicated thread to handle the communication with the VS-S while the robot's main thread executes the robot step procedures.

The singleton pattern seems to fit perfectly the use case of the VS-C. Designing the VS-C as a singleton, guarantees the component to be instantiated at most once. More precisely, the first element calling for the instantiation of the VS-C causes the execution of the constructor method, while all the further calls will return a reference to the existing instance. Such a mechanism allows an undetermined number of virtual sensors to have a reference of the same VS-C instance independently to each other, and without a central entity that creates the single VS-C instance and distributes its references. Moreover, the VS-C constructor method is guaranteed to execute at most once, and it can be exploited to spawn the thread dedicated to the VS-C life cycle. As soon as the single VS-C instance is created, the new thread is launched from within the constructor and the VS-C life cycle exposed in Figure 3.18 is associated to it as its start routine.

The VS-C connects to the VS-S using the address and port given as attributes in the XML configuration file. The XML file is parsed in the initialisation method of the Arena Tracking System Real E-Puck, and the VS-S address is searched in the attributes *vss_host* and *vss_port* of the *experiment* node. The VS-S address is then set in the VS-C through the method *SetServerAddressAndPort*. In the VS-C initialisation method, the client waits for the server address to be filled by the ATS Real E-Puck and then connects via TCP protocol to the VS-S. When the VS-S receives the connection request, it immediately sends the VST. The VS-C is ready to receive and deserialise the hash table, creating a local version of the VST. The local Robot Virtual Sensor Table, or R-VST, is used by the VS-C to split the byte buffer of the virtual sensor data coming from the VS-S according to the virtual sensors data size stored in the R-VST.

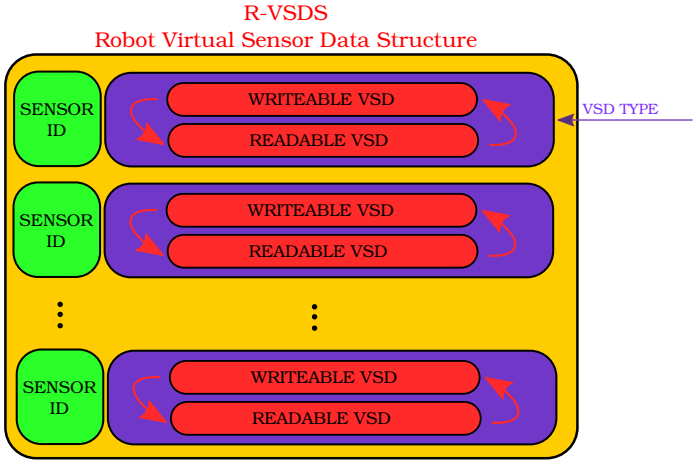

Figure 3.19: Robot Virtual Sensor Data Structure.

After that, the VS-C enters the outer loop of its execution, waiting for the start command from the VS-S. When the researcher presses the play button on the ARGoS GUI, the start command is sent to all the robots and their VS-C triggers the execution of the control software running on the robot's main

thread. At the same time, the VS-S enters the inner loop where it repeatedly waits for virtual sensor data until the the experiment run is over. The virtual sensor data is deserialised by the VS-C and the Robot Virtual Sensor Data Structure is created. The Robot Virtual Sensor Data Structure, or R-VSDS, contains the same information of the server side R-VSD, but in a suitable format to be accessed by single virtual sensors rather than the whole robot. While the R-VSD is a byte buffer containing serialised couple of sensor id and sensor data, in the R-VSDS the couples are chunked from the byte buffer and reordered to create a more convenient entry list indexed with the sensor ID (see Figure 3.19). The serialised sensor data is the same portion of byte array situated after the sensor ID in the R-VSD. In fact, the serialised sensor data is sensor specific and its dimension is stored in the R-VST together with the corresponding sensor ID. Moreover, the R-VSDS is a data structure shared by two threads: the thread that hosts the VS-C and the robot's main thread. In fact, in the virtual sensors update phase of the robot step, all the virtual sensors are asked to read the R-VSDS and set the sensor value in their reading structure. The exclusive access to the R-VSDS is protected once again by a double buffer mechanism. Similarly to the AS-DS, the critic data is duplicated: the writeable version is accessed only by the VS-C, while the readable one is obtained in read only mode by the virtual sensors. Each time the VS-C completes the update of the R-VSDS, it swaps the double buffer using a mutex to deny virtual sensor readings for this short amount of time. When the run expires, the VS-C receives the stop signal and returns to the initial state of the loop waiting, for the new run to start.

During the loop, the VS-C is constantly listening for socket exceptions. In case the researcher quits ARGoS GUI after the last run, the VS-S shuts down as well. The VS-S shut down raises an exception to the sockets connected to it. The exception is handled on the VS-C as an abort signal, and when caught, the VS-C must exit the execution loop and terminate.

3.4.2 Arena Tracking System Real E-Puck

Similarly to the ARGoS simulator, the robots' main program is responsible for executing in order all the procedures required to accomplish a robot step. The entity assigned to implement all the procedures is the Arena Tracking System Real E-Puck, or ATS-RE. The ATS-RE is the implementation for the ATS-VSP of the real robot entity. The ATS-RE must handle all the real sensors and actuators of the real E-Puck it is modelling. In this case, the ATS-RE is only a specialisation of the real E-Puck component in the ARGoS E-Puck plugin, since the robot architecture is still unchanged, but the new real robot entity must handle also the virtual sensors. Figure 3.20 shows the add-on features of the ATS version of the real E-Puck.

To the list of I2C sensors and of serial sensor, the ATS-RE adds the list of virtual sensors. The difference between I2C sensors and serial sensors lies in the fact that the E-Pucks used for the ATS-VSP are extensions of the basic E-Puck architecture. As stated in Chapeter 1.3, the extensions include a set of ground sensors, the range and bearing sensor/actuator, and the Linux board. The two extra sensors are directly interfaced with the Linux board through an I2C bus, in order to bypass the PIC micro controller. All the other sensors instead are interfaced to the PIC micro controller by default, and their data must be gathered from there to the Linux board at each robot step.

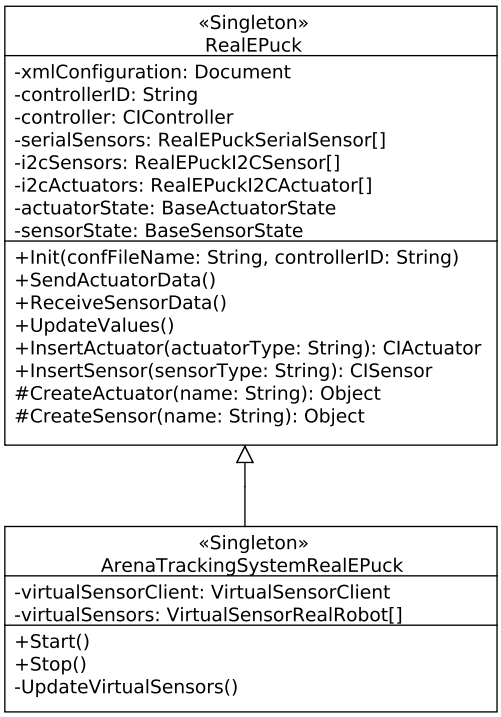

Figure 3.20: Arena Tracking System Real E-Puck class diagram.

Figure 3.21 exemplifies the robot states for an execution step. After the initialisation phase, the robot waits still for the start signal. The busy waiting is performed by a cyclic access to a VS-C flag declaring the active or inactive status of the ARGoS run. The start signal is delivered by the VS-S to the VS-C when the researcher presses the play button on the ARGoS GUI. In the meantime, the VS-C is blocked on the the start signal waiting. When it receives that signal, the VS-C sets the flag on which the ATS-RE is constantly checking for the condition. Since the flag is shared between two processes, the access must be guaranteed mutually exclusive. Thanks to this mechanism, the action of the researcher on the ARGoS GUI triggers all the components in a chain all the way to the real robots, enabling their execution.

When the start signal arrives to the ATS-RE, the thread switches to the next state, the receive sensor data state. In this state, the robot accesses the real sensors connected to the I2C bus to store their values in the real sensors' reading data structure. In the next state, the update values state, the same action is taken for the sensors directly connected to the PIC micro controller. Finally, the virtual sensors updating state takes place. Every instantiated virtual sensor executes its *UpdateValues* method, reading the sensor value from the R-VSDS, deserialising and storing it in the proper reading data structure (see Section 3.4.3).

When I2C sensors, serial sensors and virtual sensors are updated, the ATS-RE performs the control step, where the user defined control logic is executed and the decisions on the actuators are taken. After the control computation, the actuator data emerged in the control step are sent to the actuators through

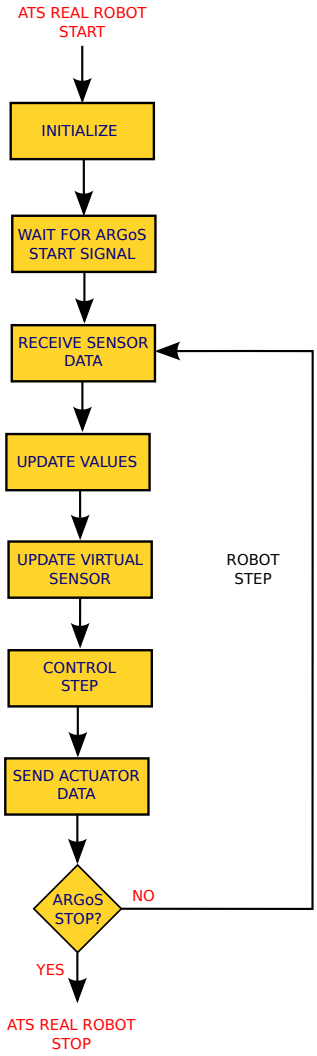

Figure 3.21: Robot main work flow.

the I2C bus in case of the range and bearing actuator, or through the PIC micro controller for all the other actuators.

3.4.3 Virtual Sensors Real Robot Module

The virtual sensor real robot modules are the final users of the data flow started with the image processing in the ATS. These components are the counterparts of the virtual sensor simulator modules, and must be implemented as well by the virtual sensor designer, because in ARGoS each virtual sensor is composed by a couple of simulator and real robot modules.

Like the simulator module, a virtual sensor real module must implement some interfaces: the specific virtual sensor control interface, and the generic virtual sensor real robot module interface. The specific virtual sensor interface

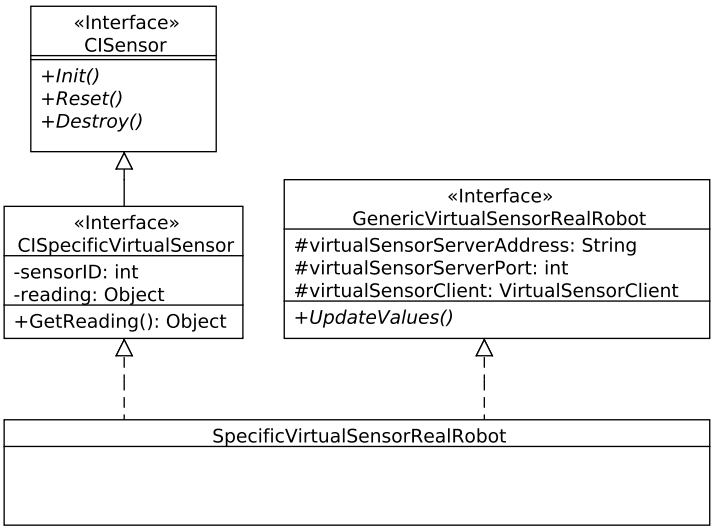

Figure 3.22: Virtual Sensor Real Robot Module class diagram.

is the same interface implemented by the simulator module and described in Section 3.3.4. As explained above, this interface gives the component the right to be a *sensor*.

The generic virtual sensor real robot module is a single interface that unifies the semantic of both generic virtual sensor simulator module and simulated sensor interfaces for the simulator module. A component that implements the generic virtual sensor real robot interface gains the status of *virtual* component targeted for *real robots*. The virtual status is given by the reference to the VS-C inherited by default from all the components implementing the interface. The real robot module status is given by the abstract method *UpdateValues*, which is meant to be implemented following the semantic of the alike method of all serial sensors. In particular, the *UpdateValues* method must update the reading data structure that is accessed by the control software through the *GetReading* method of the specific control interface. The general approach is to access the R-VSDS with the sensor ID and retrieve the corresponding readable data buffer. The method then must deserialise the single sensor data buffer and store the values in the reading data structure inherited from the specific virtual sensor control interface. The deserialisation must be implemented according to the serialisation algorithm of the virtual sensor simulator module.

The architecture developed allows the virtual sensor designer to create any kind of virtual sensor needed. Great flexibility is a major feature in ARGoS, and the virtual sensing extension is designed keeping flexibility as a cornerstone. In the ATS-VSP, flexibility is achieved thanks to the freedom left to the virtual sensor designer on the main design choices. The methods *Update* for the VS-SM and *UpdateValues* for the VS-RRM are the key points for the design of the virtual sensor and they are completely implemented by the virtual sensor designer. The specific virtual sensor control interface and the Virtual Sensor Register also contribute to the system flexibility, allowing for the new components to be part of the system like all the other sensors.

Reusability is another feature promoted by the ARGoS plugins architecture,

and the ATS-VSP follows the same guidelines. In the ATS-VSP, the whole E-Puck plugin is reused with the sole exception of the real robot entity, the ATS-RE, that is actually an extension of the Real E-Puck of the E-Puck plugin. The virtual sensor designer can exploit ARGoS reusability creating generic virtual sensors and reusing them to build robot specific virtual sensors. In the case the virtual sensor designer decides to virtualise a sensor that is already simulated in ARGoS, he/she can reuse the whole simulation module of the existing sensor focusing on the implementation of the only real robot module.

Given the explanation of the ATS-VSP features and purpose, the reader should now be aware of the generic architecture of the plugin, and where the virtual sensor designer is asked to operate for virtual sensors implementation. The next chapter is a guide for the virtual sensor designer meant to describe the creation of different kinds of virtual sensors step by step.

Chapter 4

Virtual Sensors implementation

This chapter is a guide for virtual sensors implementation. I am going to describe step by step the implementation of two virtual sensors: a ground virtual sensor and a light virtual sensor. These examples stand as straightforward guidelines to build any kind of virtual sensor.

The chapter is structured as follows: in Section 4.1 I demonstrate how to create a control interface for virtual sensors, in Section 4.2 I show how to implement the simulator module of a virtual sensors, while the real robot module is described in Section 4.3.

4.1 Control Interface

The control interface of a sensor is a component that provides an abstract method for accessing the sensor value, called *GetReading*, hiding the method implementation to the caller. The control interface defines the data structure of the sensor reading, but demands the implementation logic to the classes that support the interface. The classes that implement a sensor interface are the sensor's simulator module and the sensor's real robot module. This mechanism achieves control software portability from simulation to real robots. In fact, in the control software every sensor is referred through its control interface, and the execution of the method is demanded to the proper module instantiated by the system.

Virtual sensors are an extended version of regular sensors. That means, all the features of a regular sensor must be provided in the virtual sensor too. Creating the virtual sensor control interface is the first step for virtual sensors implementation. The virtual sensor control interface must extend the generic sensor control interface and define the abstract method *GetReading*. In case the virtual sensor reading is an abstract data type, the virtual sensor control interface must define also its data structure. Figure 4.1 shows the control interfaces of three virtual sensors: light virtual sensor, pollutant virtual sensor, and RGB ground virtual sensor. The RGB ground virtual sensor data is not a standard data type, but a *Color* data type. The Color data structure must be defined or imported in the control interface.

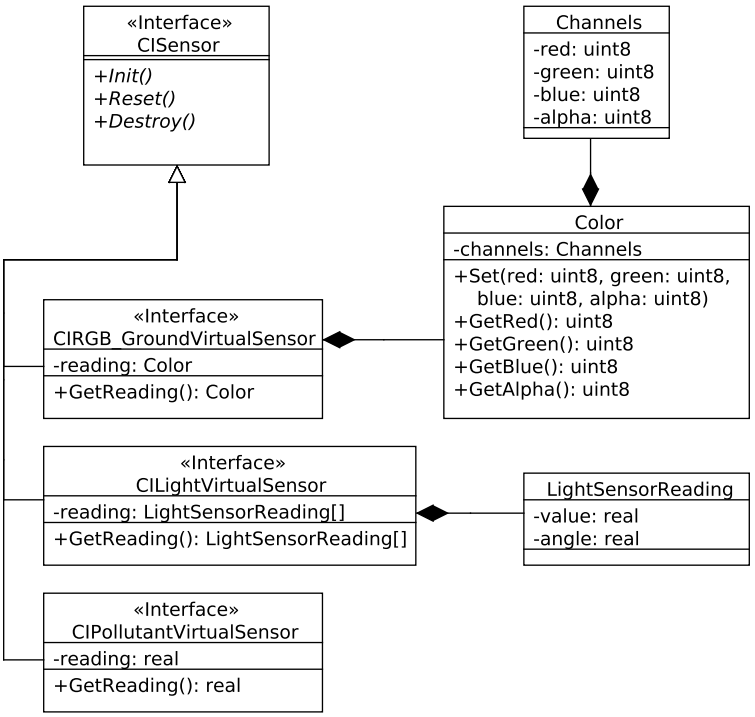

Figure 4.1: Virtual Sensor Control Interface class diagram.

The creation of a control interface is not enough for the component to be perceived as a sensor by the system. The other needed operation is creating the virtual sensor control interface entry in the VSR. In the VSR, the virtual sensor designer must add an entry formed by a pair: a string for the virtual sensor unique name, and an integer for the virtual sensor unique ID (see Figure 4.2). When the control interface has been implemented and the VSR has been updated, the virtual sensor control interface is set, and the virtual sensor designer can move on to the next step: the virtual sensor simulator module implementation.

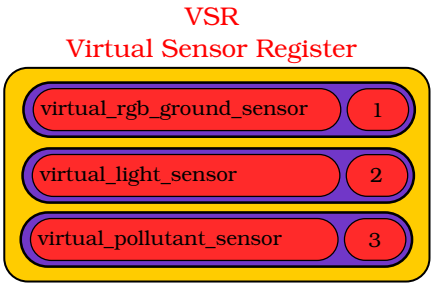

Figure 4.2: Virtual Sensor Register. Each entry includes the virtual sensor name and the virtual sensor ID.

4.2 Simulator

The simulator module of a virtual sensor is the part of the virtual sensor executed by ARGoS. This module extends the generic virtual sensor interface, the simulated sensor interface, and the control interface implemented in the previous section. The task of this component is to compute the virtual sensor reading in the *Update* method. The power of this component is the possibility to access all the information regarding the current step of the simulation. Once the virtual reading is calculated, the virtual sensor simulator module must serialize the data structure and include the buffer in the VS-DS. The remainder of this section shows how to implement the virtual sensor simulator module for the RGB ground virtual sensor, the light virtual sensor and the pollutant virtual sensor.

4.2.1 Ground Virtual Sensor Simulator Module

The ground sensor is a sensor able to determine the colour of the floor underneath the robot. The colour of the floor can be used in the experiments to model the location of a resource or to define an area that is different from the rest of the arena. For instance, an experiment on robot aggregation can use floor colour to distinguish the area where the robots must aggregate. In a foraging experiment instead, several areas of different colours can be used to represent the nest and the food locations. However, the ground sensor installed on the E-Puck is able to recognize only grey scale colours, preventing the researcher to use colours to differentiate the areas. In reality, the E-Puck ground sensor values must be strongly quantisated to avoid erroneous readings. The result is that the ground sensor is able to determine only three colours: white, black, and only one shade of grey.

Virtualizing the ground sensor allows the researcher to design a virtual floor for the arena. The benefits are two. First, the floor is much easier to create and modify in the form of a jpg image, rather than a physical panel the size of the arena. Second, the ground virtual sensor is error-free, i.e., it does not need quantisation and all 255 values of the grey scale are correctly distinguished.

Another enhancement brought by the virtualisation of the ground sensor, is that the ground virtual sensor can perform RGB detection. In this section, I am going to show the realizations of the RGB ground virtual sensor, for the simulator module.

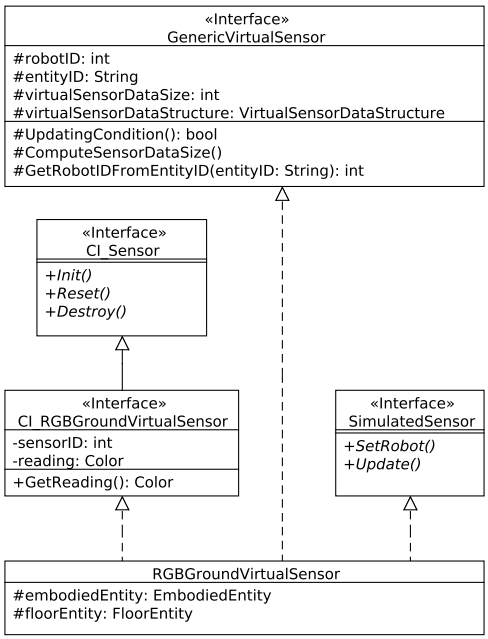

Figure 4.3: Class diagram of the RGB ground virtual sensor for simulator.

The RGB ground virtual sensor simulator modules inherits the attributes and methods of the implemented interface, and must provide an implementation logic for the abstract methods of the interfaces. The component inherits four attributes from the generic virtual sensor interface: and two attributes from the control interface. The attributes inherited from the control interface are the sensor reading, an attribute of Colour type, and the sensor ID, the unique identifier as defined in the VSR. The attributes inherited from the generic virtual sensor interface are the robot ID, the entity ID, a reference to the VS-DS and an integer that stores the size in bytes of the virtual sensor reading data. The robot ID is an integer that represents the robot to the VS-S, while entity ID is a string that represents the robot in ARGoS. The virtual sensor data size must be computed according to the data type of the reading. The computation of this value must be implemented in a procedure called ComputeSensorDataSize and executed in the init method. The RGB ground virtual sensor reading data type is composed by four variables of typr uint8, as shown in Figure 4.1. Therefore, the ComputeSensorDataSize method simply returns the value 32. The other method of the generic virtual sensor interface that must be implemented by the RGB ground virtual sensor is the method UpdatingCondition. This method returns a boolean value that represents the condition under which the reading can be appended in the VS-DS. In the RGB ground virtual sensor, the reading value is appended to the VS-DS and sent to the robot only if it is different from the last reading sent. To do this, a variable of type Color is required to store the last sent reading. The method returns the inverted result of the comparison between the current reading and the last sent reading value.

```
bool RGBGroundVirtualSensor::UpdatingCondition()
```

```
{
    return lastSentReading!=reading;
}
```

The abstract methods left to be implemented are: Init, Reset, Destroy, SetRobot and Update. The Init method in the RGB ground virtual sensor is used to set the sensor data size value, insert the RGB ground virtual sensor entry in the VST, and initialise the reading value The sensor data size is calculated by the ComputeSensorDataSize method, and the initialisation of the reading structure is performed by the Update function. The Init method is therefore implemented as follows:

```
void RGBGroundVirtualSensor::Init(ConfigurationNode& tree)
{
   CI_RGBGroundVirtualSensor(tree);
   ComputeSensorDataSize();
   sensorID =
       VirtualSensorRegister::GetVirtualSensorId("GroundVirtualSensor");
   if (!virtualSensorDataStructure.ExistsSensorInVST(sensorID)) {
      virtualSensorDataStructure
        .AddVirtualSensorEntry(sensorID, virtualSensorDataSize);
   }
   Update();
}
```

The Reset and Destroy method are left empty, because there is no need to reset the value of the reading or to free allocated memory.

The SetRobot method is used to bind the two attribute entities *embodiedEntity*, and *floorEntity* with the corresponding robot entity components. In addiction, this method sets the entity ID and the robot ID exploiting the entity object given as parameter. The SetRobot method is implemented as follows:

```
void RGBGroundVirtualSensor::SetRobot(ComposableEntity& entity)
{
   entityID = entity.GetId();
   robotID = GetRobotIDFromEntityID(entityID);
   embodiedEntity = &(entity.GetComponent<EmbodiedEntity>("body"));
   Space& space = Simulator::GetInstance().GetSpace();
   floorEntity = &space.GetFloorEntity();
}
```

The Update method is used to set the new reading value. In the RGB ground virtual sensor the reading value is provided by the floor entity. The Update method is implemented as follows:

```
void RGBGroundVirtualSensor::Update()
{
   Vector3D entityPosition = embodiedEntity->GetPosition();
   reading = floorEntity->GetColorAtPoint(entityPosition.GetX(),
       entityPosition.GetY());

   if(UpdatingCondition()) {
      lastSentReading = reading;
```

```
        int bufferLength = virtualSensorDataSize + 1;
        char byteDataBuffer[bufferLength];
        byteDataBuffer[0] = sensorID;
        byteDataBuffer[1] = reading.GetAlpha();
        byteDataBuffer[2] = reading.GetRed();
        byteDataBuffer[3] = reading.GetGreen();
        byteDataBuffer[4] = reading.GetBlue();
        virtualSensorDataStructure.AppendVirtualSensorData(byteDataBuffer,
            bufferLength, robotID);
    }
}
```

The implementation of the Update method is the last step to produce the simulator module of a virtual sensor. In the next subsection the procedure will be repeated to build a light virtual sensor simulator module.

4.2.2 Light Virtual Sensor Simulator Module

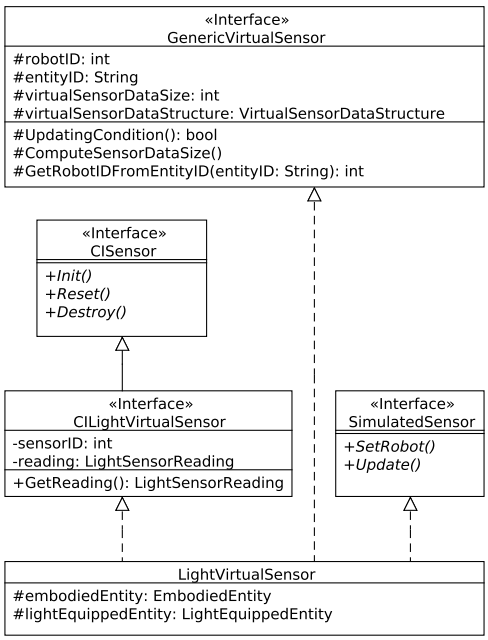

Figure 4.4: Class diagram of the light virtual sensor for simulator.

The virtualisation of the light sensor allows the researcher to carry out experiments using light sources that may not be physically used in the lab for several reasons. The researcher can use any desired number of light sources without worrying about electric powering, positioning in the space, undesired occlusions due to arena furniture external to the experiment or people walking in the arena, undesired light reflection on the walls or the floor of the arena, or simply the unavailability of a proper dimmer for the light sources.

The E-Puck light sensor is actually a set of eight light sensors disposed in a ring on the border of the E-Puck robots. The angle of each sensor in the ring

is defined in the light sensor equipped entity, and can be retrieved through a specific accessor. The generic light sensor reading is an array of real values that stores the normalized value of the light intensity perceived by that sensor. The simple light intensity is not a complete information for the control software, therefore the light virtual sensor reading is integrated with the angle position of each sensor. The angle is constant and must be set in the Init method. The Init method must call the ComputeSensorDataSize method and insert the virtual sensor entry in the VST as well.

```
void LightVirtualSensor::Init(ConfigurationNode& tree)
{
   CI_LightVirtualSensor::Init(tree);
    ComputeSensorDataSize();
    sensorID =
        VirtualSensorRegister::GetVirtualSensorId("LightVirtualSensor")
    if (!virtualSensorDataStructure.ExistsSensorInVST(sensorID)) {
        virtualSensorDataStructure
            .AddVirtualSensorEntry(sensorID, virtualSensorDataSize);
    }
    reading.resize(lightEquippedEntity->GetNumSensors());
    for (int i=0; i<reading.size(); i++) {
        Vector3D direction = lightEquippedEntity->GetSensor(i).Direction;
        reading[i].Angle = direction.GetXAngle().GetValue();
    }
    Update();
}
```

The light virtual sensor reading is a set of LightSensorReading data structure. The LightSensorReading data structure contains two real values: the light intensity value and the sensor angle in the ring. The light virtual sensor data size is therefore the size of the two real values times the number of sensors.

```
void LightVirtualSensor::ComputeSensorDataSize()
{
    int lightSensorReadingSize = 2* sizeof(Real);
    virtualSensorDataSize = lightSensorReadingSize *
        lightEquippedEntity->GetNumSensors();
}
```

The SetRobot method must bind the entities used by the virtual sensor to those instantiated as part of the robot entity.

```
void LightVirtualSensor::SetRobot(ComposableEntity& entity)
{
    entityID = entity.GetId();
    robotID = GetRobotIDFromEntityID(entityID);
    embodiedEntity = &(entity.GetComponent<EmbodiedEntity>("body"));
    lightEquippedEntity =
        &(entity.GetComponent<LightEquippedEntity>("light_sensors"));
}
```

As for the RGB ground virtual sensor, the methods Reset and Destroy are not implemented. The method UpdatingCondition instead returns true, since it

is very rare that the light intensity does not change for any of the eight sensors. The politics used in this case is to send the update light virtual sensor data at every timestep.

The Update method of the light virtual sensor is a bit more complicated than the Update method of the RGB ground virtual sensor. In fact, the method must repeat a complex procedure for any of the light sensors on the ring. The procedure consists in iterating all the simulated light sources, check for possible occlusions, and if none is detected then calculate the light intensity for the given light source on the given light sensor.

The code of the Update method is shown in the snippet below.

```
void LightVirtualSensor::Update()
{
   /* Erase readings */
   for(int i = 0; i < reading.size(); ++i) reading[i].value = 0.0f;
   /* Ray used for scanning the environment for obstacles */
   Ray3D scanningRay;
   Vector3D rayStart;
   Vector3D sensorToLight;
   /* Buffers to contain data about the intersection */
   EmbodiedEntityIntersectionItem intersection;
   /* Get the map of light entities */
   Space& space = Simulator::GetInstance().GetSpace();
   Space::MapPerTypePerId::iterator itLights =
       space.GetEntityMapPerTypePerId().find("light");
   if (itLights != space.GetEntityMapPerTypePerId().end()) {
      Space::MapPerType& mapLights = itLights->second;
      /* Go through the sensors */
      for(int i = 0; i < reading.size(); ++i) {
         /* Set ray start */
         rayStart = lightEquippedEntity->GetSensor(i).Position;
         rayStart.Rotate(embodiedEntity->GetOrientation());
         rayStart += embodiedEntity->GetPosition();
         /* Go through all the light entities */
         for(Space::MapPerType::iterator it = mapLights.begin();
            it != mapLights.end(); ++it)
         {
            /* Get a reference to the light */
            LightEntity& light = *any_cast<LightEntity*>(it->second);
            /* Consider the light only if it has non zero intensity */
            if(light.GetIntensity() > 0.0f) {
               /* Set ray end to light position */
               scanningRay.Set(rayStart, light.GetPosition());
               /* Check occlusions */
               if(!GetClosestEmbodiedEntityIntersectedByRay(intersection,
                    scanningRay))
               {
                  /* No occlusion, the light is visibile */
                  /* Calculate reading */
                  scanningRay.ToVector(sensorToLight);
                  reading[i].value +=
                       CalculateReading(sensorToLight.Length(),
                       light.GetIntensity());
```

```
            }
          }
        }
      }
      /* Send readings to the robot */
      int bufferLength = virtualSensorDataSize + 1;
      char byteDataBuffer[bufferLength];
      int indexBuffer = 0;
      byteDataBuffer[unIndexBuffer] = sensorID;
      indexBuffer = indexBuffer + 1;
      std::vector<Real>::const_iterator itRealVector;
      /* For each reading add it in byte form to the byte data buffer */
      for (itRealVector = reading.begin(); itRealVector !=
          reading.end(); itRealVector++)
      {
         memcpy(byteDataBuffer + indexBuffer,
             &((*itRealVector).value),sizeof(Real));
         indexBuffer = indexBuffer + sizeof(Real);
         memcpy(byteDataBuffer + indexBuffer,
             &((*itRealVector).angle),sizeof(Real));
         indexBuffer = indexBuffer + sizeof(Real);
      }
      /* Append the Virtual Sensor byte data buffer in the Virtual
          Sensor Data Struct */
      virtualSensorDataStructure.AppendVirtualSensorData(byteDataBuffer,
          bufferLength, robotID);
   }
}
```

The procedure CalculateReading used in the Update method is defined as follows.

```
void LightVirtualSensor::CalculateReading(Real distance, Real intensity)
{
   return (intensity * intensity) / (distance * distance);
}
```

4.2.3 Pollutant Virtual Sensor Simulator Module

The pollutant virtual sensor is a virtual sensor that is able to perceive the environmental pollution in the robot location. The implementation of the pollutant virtual sensor assumes that the pollution is radiated from a pollutant entity in a certain direction, and it spreads in the environment within a diffusion cone. The pollutant entity, the radiation angle, the width of the pollutant cone, and the maximum distance from the pollutant entity are parameters defined at the beginning of the experiment by the researcher in the ARGoS XML configuration file. These parameters are parsed in the Init method.

```
void PollutantVirtualSensor::Init(ConfigurationNode& tree)
{
   CI_PollutantVirtualSensor::Init(tree);
   ComputeSensorDataSize();
```

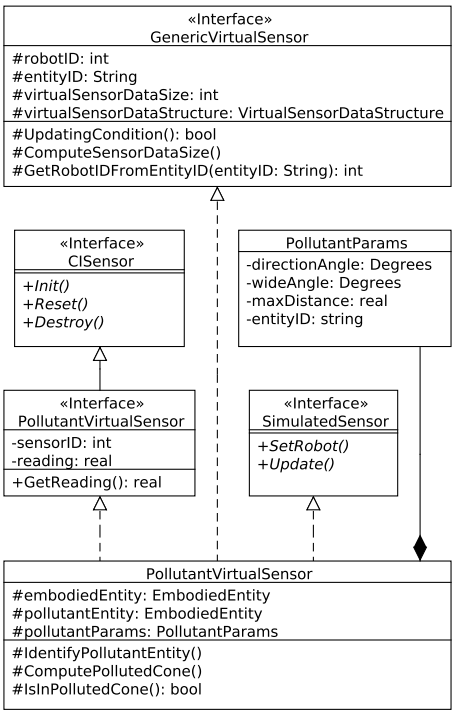

Figure 4.5: Class diagram of the pollutant virtual sensor for simulator.

```
   sensorID =
       VirtualSensorRegister::GetVirtualSensorId("PollutantVirtualSensor");
   /* Parse XML file */
   GetNodeAttributeOrDefault(tree, "direction_angle",
       pollutantParams.directionAngle, Degrees(0));
   GetNodeAttributeOrDefault(tree, "wide_angle",
       pollutantParams.wideAngle, Degrees(90));
   GetNodeAttributeOrDefault(tree, "max_distance",
       pollutantParams.maxDistance, 1.0);
   GetNodeAttributeOrDefault(tree, "pollutant_entity",
       pollutantParams.entityID, "epuck_22_1");
    if (!virtualSensorDataStructure.IsSensorAlreadyInTable(sensorID)) {
        virtualSensorDataStructure.AddVirtualSensorEntry(sensorID,
            virtualSensorDataSize);
    }
    Update();
}
```

Since the pollutant virtual sensor reading is a real value, the method ComputeSensorDataSize returns the size of real type.

```
void PollutantVirtualSensor::ComputeSensorDataSize()
{
   return sizeof(Real);
}
```

The SetRobot method must bind not only the sensor's entities to the robot ones, but also the pollutant entity to its robot entity. The method IdentifyPollutantEntity is the procedure demanded to cycle the entities in the ARGoS space and find the one defined by the ID stored in pollutantParams.entityID. When the pollutant entity is retrieved, the pollutant cone can be calculated by the procedure ComputePollutedCone.

```
void PollutantVirtualSensor::SetRobot(ComposableEntity& entity)
{
   entityID = entity.GetId();
   robotID = GetRobotIDFromEntityID(entityID);
   embodiedEntity = &(entity.GetComponent<EmbodiedEntity>("body"));
   IdentifyPollutantEntity();
   ComputePollutedCone();
}
```

```
void PollutantVirtualSensor::IdentifyPollutantEntity()
{
   try{
      /* Get the map of all epucks from the space */
      Space& space = Simulator::GetInstance().GetSpace();
      Space::MapPerType& epuckMap = space.GetEntitiesByType("epuck");
      /* Go through them */
      for(Space::MapPerType::iterator it = epuckMap.begin(); it !=
          epuckMap.end(); ++it) {
         EPuckEntity* epuckEntity = any_cast<EPuckEntity*>(it->second);
         if (epuckEntity->GetId() == pollutantParams.entityID){
            pollutantEntity = &(epuckEntity->GetEmbodiedEntity());
            break;
         }
      }
   }
   catch (CARGoSException e){}
}
```

The procedure ComputePollutedCone calculates the determinant of the triangle matrix of the pollutant cone.

```
void PollutantVirtualSensor::ComputePollutedCone(){
   if (pollutantEntity != NULL){
      /* Compute the triangle of influence */
      Vector3D sourcePos3 = pollutantEntity->GetPosition();
      vertex1.Set(sourcePos3.GetX(), sourcePos3.GetY());
      vertex2.FromPolarCoordinates(pollutantParams.maxDistance,
          ToRadians(-pollutantParams.wideAngle/2));
      vertex2.Rotate(ToRadians(pollutantParams.directionAngle));
      vertex2 += vertex1;
      vertex3.FromPolarCoordinates(pollutantParams.maxDistance,
          ToRadians(+pollutantParams.wideAngle/2));
      vertex3.Rotate(ToRadians(pollutantParams.directionAngle));
      vertex3 += vertex1;
```

```
      /* Compute the determinant of Triangle Matrix (y2 - y3)(x1 - x3) +
          (x3 - x2)(y1 - y3) */
      triangleMatrixDet = (vertex2.GetY() -
          vertex3.GetY())*(vertex1.GetX() - vertex3.GetX()) +
           (vertex3.GetX() - vertex2.GetX())*(vertex1.GetY() -
               vertex3.GetY());
   }
}
```

When the pollutant cone is computed, it is possible to calculate whether or not the robot is located within the cone.

```
bool PollutantVirtualSensor::IsInPollutedCone()
{
   /* Get robot position */
   Vector3D& robotPos3D = embodiedEntity->GetPosition();
   Vector2D robotPos(robotPos3D.GetX(),robotPos3D.GetY());
   Real x = robotPos.GetX();
   Real y = robotPos.GetY();
   /* r1 = (y2 - y3)(x - x3) + (x3 - x2)(y - y3) all divided by MatDet */
   robotPos.SetX( ((vertex2.GetY() - vertex3.GetY())*(x -
       vertex3.GetX()) + (vertex3.GetX() - vertex2.GetX())*(y -
       vertex3.GetY()))/triangleMatrixDet );
   /* r2 = (y3 - y1)(x - x3) + (x1 - x3)(y - y3) all divided by MatDet */
   robotPos.SetY( ((vertex3.GetY() - vertex1.GetY())*(x -
       vertex3.GetX()) + (vertex1.GetX() - vertex3.GetX())*(y -
       vertex3.GetY()))/triangleMatrixDet );
   /* if robot inside polluted triangle set Value to 1 --- that is 0 <=
       r1 <= 1 and 0 <= r2 <= 1 and r1 + r2 <= 1*/
   if (robotPos.GetX() >= 0 && robotPos.GetX() <= 1 && robotPos.GetY()
       >= 0 && robotPos.GetY() <= 1 && (robotPos.GetX()+robotPos.GetY())
       <= 1)
   {
      return true;
   }
   return false;
}
```

Finally, the Update method puts together all the procedures to update the reading value and appends the serialized virtual sensor data.

```
void PollutantVirtualSensor::Update()
{
   /* Default virtual sensor Value is ZERO */
   reading = 0;
   if (pollutedEntity != NULL){
      ComputePollutedCone();
      if(IsInPollutedCone) {
         reading = 1.0f;
      }
   }
   int bufferLength = virtualSensorDataSize + 1;
   char byteDataBuffer[bufferLength];
   byteDataBuffer[0] = sensorID;
```

```
    memcpy(byteDataBuffer + 1, &reading, sizeof(Real));
    virtualSensorDataStructure.AppendVirtualSensorData(byteDataBuffer,
        bufferLength, robotID);
}
```

At this point, the pollutant virtual sensor simulator module is completed. The next session shows how to implement the E-Puck module of the RGB ground virtual sensor, the light virtual sensor and the pollutant virtual sensor.

4.3 Real Robot

In the ATS-RE file, the virtual sensor designer must modify the *InsertSensor* method, adding the possibility for the ATS-RE to recognise the new virtual sensor when parsing the XML configuration file. The following is a code snippet from the ATS-RE InsertSensor method.

```
CI_Sensor* ATSRealEPuck::InsertSensor(string sensor_name) {
    if (sensor_name == "virtual_rgb_ground_sensor") {
        RealRGBGroundVirtualSensor* virtualRGBGroundSensor =
            CreateSensor<RealRGBGroundVirtualSensor>(sensor_name);
        virtualSensors.push_back(virtualRGBGroundSensor);
        return virtualRGBGroundSensor;
    }
    if (sensor_name == "virtual_light_sensor") {
        RealLightVirtualSensor* virtualLightSensor =
            CreateSensor<RealLightVirtualSensor>(sensor_name);
        virtualSensors.push_back(virtualLightSensor);
        return virtualLightSensor;
    }
    if (sensor_name == "virtual_pollutant_sensor") {
        RealPollutantVirtualSensor* virtualPollutantSensor =
            CreateSensor<RealPollutantVirtualSensor>(sensor_name);
        virtualSensors.push_back(virtualPollutantSensor);
        return virtualPollutantSensor;
    }
    else {
        return RealEPuck::InsertSensor(sensor_name);
    }
}
```

Figure 4.6 shows the class diagram of the three implemented virtual sensor real robot modules. Every virtual sensor real robot module must implement four abstract methods: Init, Reset, Destroy and UpdateValues. The remainder of this section shows how to implement the virtual sensor E-Puck module for the RGB ground virtual sensor, the light virtual sensor and the pollutant virtual sensor.

4.3.1 Virtual Sensor Real Robot Modules implementation

The E-Puck module of the virtual sensors is the component that retrieves the sensor's value from the R-VSDS and sets the reading value. In the Init method, every virtual sensor E-Puck module must parse the XML configuration file and

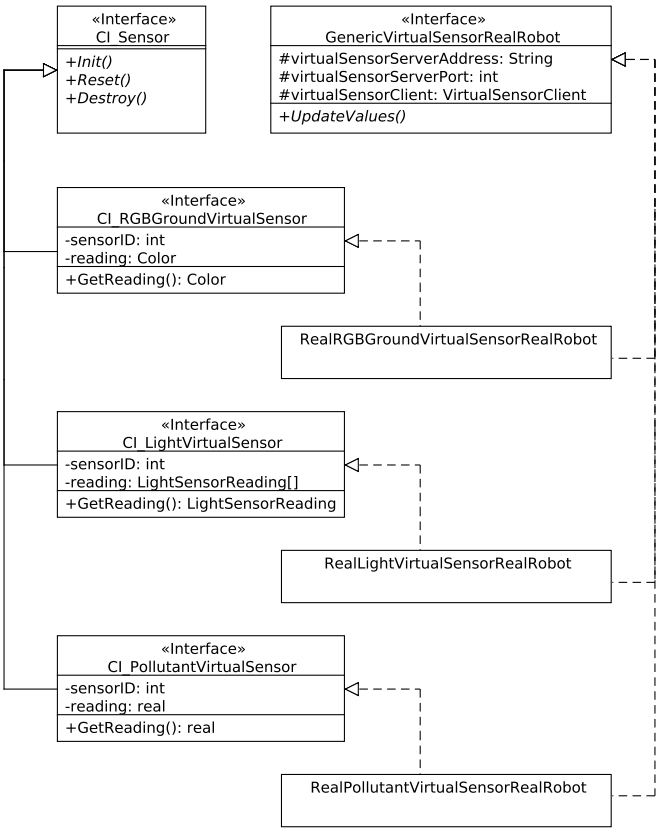

Figure 4.6: Class diagram of the virtual sensor real robot modules.

notify the VS-S host and port to the VS-C. This operation is performed by every virtual sensor, however only the first time the operation is effective. The code of the Init method of the RGB ground virtual sensor is as follows. The Init method of the other virtual sensors only differs for the string parameter passed to the GetVirtualSensorId method.

```
void RealRGBGroundVirtualSensor::Init(ConfigurationNode& node)
{
   sensorID(VirtualSensorRegister::GetVirtualSensorId("GroundVirtualSensor"));
   /* Parse XML configuration file and get Virtual Sensor Server Host
       and Port */
   GetNodeAttributeOrDefault<std::string>(node, "vss_host",
       virtualSensorServerAddress, virtualSensorServerAddress);
   GetNodeAttributeOrDefault<uint32_t>(node, "vss_port",
       virtualSensorServerPort, virtualSensorServerPort);
   /* Set the user defined VSS Host and Port in the Virtual Sensor
       Client */
   virtualSensorClient.SetServerAddresAndPort(virtualSensorServerAddress,
       virtualSensorServerPort);
}
```

The methods Reset and Destroy are implemented empty in the all the implemented virtual sensor E-Puck module, while the method UpdateValues deserialises the reading in the R-VSDS and store its value in the reading data structure. The code of the UpdateValues method for the RGB ground virtual sensor is shown below.

```
void RealRGBGroundVirtualSensor::UpdateValues()
{
       int bufferLength = sizeof(reading);
       char byteDataBuffer[bufferLength];
       virtualSensorClient.GetReadyData(byteDataBuffer, sensorID);
       uint8 red = byteDataBuffer[0];
       uint8 green = byteDataBuffer[1];
       uint8 blue = byteDataBuffer[2];
       uint8 alpha = byteDataBuffer[3];
       reading.Set(red, green, blue, alpha);
}
```

The UpdateValues method of the light virtual sensor differs only for the deserialisation procedure of the reading data structure, as well as the UpdateValues method of the pollutant virtual sensor.

```
void RealLightVirtualSensor::UpdateValues()
{
       int bufferLength = sizeof(reading);
       int bufferIndex = 0;
       char byteDataBuffer[bufferLength];
       virtualSensorClient.GetReadyData(byteDataBuffer, sensorID);

       for(int i=0; i<reading.size(); i++) {
         memcpy(&reading[i].value, byteDataBuffer + bufferIndex,
             sizeof(Real));
         bufferIndex++;
         memcpy(&reading[i].angle, byteDataBuffer + bufferIndex,
             sizeof(Real));
         bufferIndex++;
       }
}
```

```
void RealPollutantVirtualSensor::UpdateValues()
{
   int bufferLength = sizeof(reading);
   char byteDataBuffer[bufferLength];
   virtualSensorClient.GetReadyData(byteDataBuffer, sensorID);
   memcpy(&reading, byteDataBuffer, sizeof(Real));
}
```

Chapter 5

Validation through real robots experiment

To validate the virtual sensing technology presented, an experiment with real robots and virtual sensors has been performed at the IRIDIA lab. The experiment has been designed for the purpose of providing a simple and easily understandable example of one application of the virtual sensing technology. The experiment involves 15 E-Pucks equipped with a pollutant virtual sensor, and a pollutant entity. The pollutant entity is detectable by the ATS through an E-Puck tag placed on its top. The tag ID used for the pollutant entity must be defined as a parameter of the pollutant virtual sensor in the XML configuration file. The pollutant entity employed is a static object. However, nothing prevent to use a movable object instead, for instance an E-Puck robot. The task of the swarm consists in moving randomly in the arena, and when a robot perceives the pollutant, it stops and lights up the red LEDs with a probability $P_{\mathrm{s}} = 0.3$. To test the swarm reaction to the environment changes, the pollutant parameters are modified at runtime, after a given number of timesteps. The expectation is that the robots start to move again, looking for the new location of the pollutant cone.

The real world scenario that this experiment is simulating can be described as pollution spill detection and recovery. The spill is released in the environment by, for instance, an industrial plant simulated by the pollutant entity. A moveable pollutant entity can model a leaking oil tanker, or a vehicle emitting exhaust gas. The pollutant cone simulates the direction and intensity of the wind, in case of air pollution. The same principle can be applied to water pollution or ground pollution. The time variance of the pollutant parameters allows to simulate the changing of the wind. The robots are assumed to feature a system of air (or water, or ground) purification, which activation is simulated by the lighting of the robots red LEDs.

Starting from the pollutant entity position, the pollutant virtual sensor of each robot computes the pollutant cone, using the pollutant parameters defined by the researcher in the XML configuration file. If the robot is located inside the pollutant cone, then it stops and switches on the pollution purification system, i.e. the red LEDs (see Figure 5.1). During the experiment, the swarm exhibits the expected behaviour. In fact, robot aggregation is achieved in the pollutant

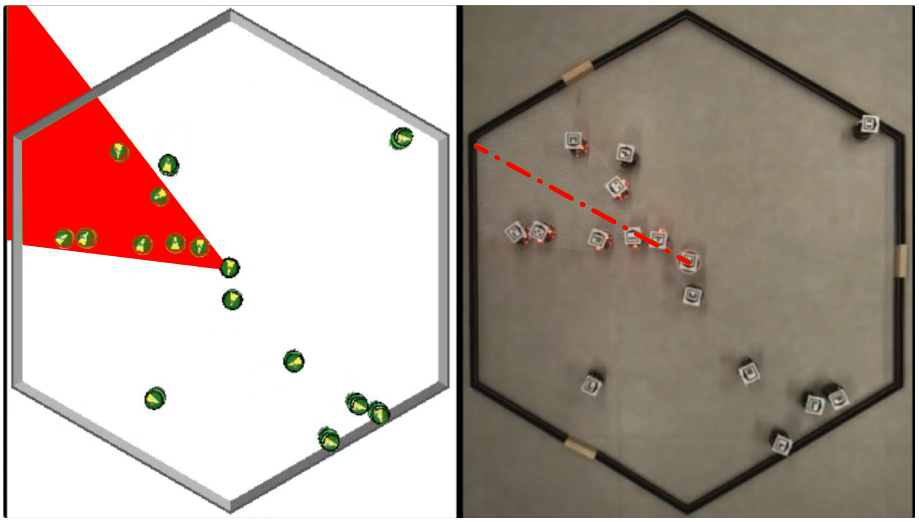

Figure 5.1: Screenshot of the experiment execution at initial environmental conditions. On the left: the simulates environment in ARGoS. On the right: a view over the arena from the camera used for tracking, with the overlay of the pollutant cone. The positions of the robots in ARGoS match the real robot ones in the arena. The pollutant entity is the object in the center of the arena that originates the pollutant cone. In the simulated environment, the pollutant cone is highlighted in red, while in the real environment the robots that perceive the pollutant thought the virtual sensor light up their red LEDs.

cone. When the pollutant cone moves due to the environment variation, the aggregated robots start to move again, achieving aggregation in the new pollutant cone location (see Figure 5.2).

One possible future enhancement to this experiment is to endow the robots with a pollutant virtual actuator (see Section 6.1). Through a pollutant virtual actuator, the robots would be able do clean the pollution in their spot modifying the virtual environment. Such an extension would allow us to study the behaviour of the swarm dealing with a dynamic environment, and test the swarm performance in carrying out a cooperative task.

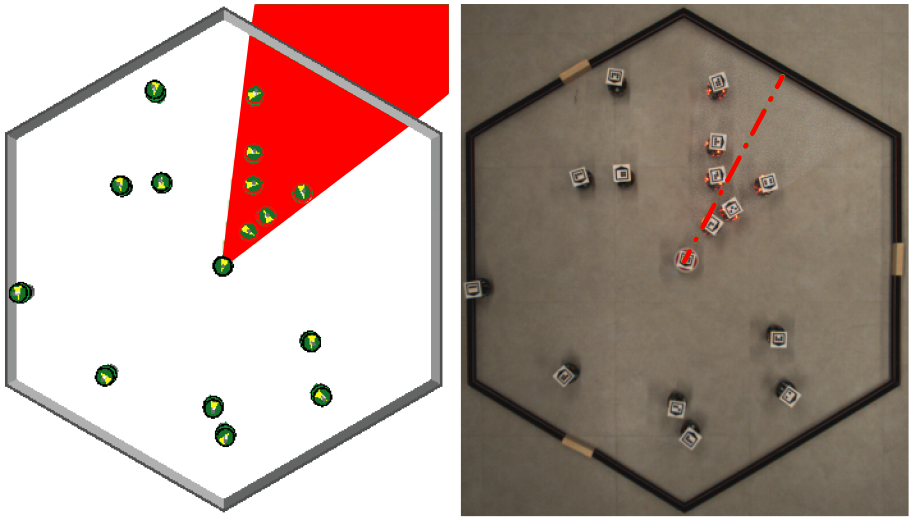

Figure 5.2: Screenshot of the experiment execution after environmental conditions variation. The robots in the real environment light up their red LEDs when they enter the new pollutant cone.

Chapter 6

Conclusions and future work

In this thesis, I presented a technology that enables virtual sensing for a swarm of E-Pucks. The system can be used for sensor prototyping and augmented reality. Sensor prototyping allows a researcher to tune a sensor by testing the swarm performance with a virtualised version of the sensor. Prototyping a sensor through virtualisation is particularly useful in swarm robotics, where the cost of hardware production and installation increases with the size of the swarm. Augmented reality for a swarm of robots allows a researcher to introduce in the environment virtual elements the are perceived by the robots. Through augmented reality the experimental environment is quick and easy to setup and modify, and it can include time variant scenarios or even physically unfeasible scenarios. The virtual sensors are easy to design and implement. I illustrate the implementation of three virtual sensors: the RGB ground virtual sensor, the light virtual sensor and the pollutant virtual sensor. Experiments involving virtual sensors are hybrid experiments between purely simulated and purely real experiments. Hybrid experiments are easier to setup and carry out than purely real experiments, and at the same time they are more accurate than the purely simulated ones. The functioning of the system has been showcased through a hybrid experiment involving 15 robots using the pollutant virtual sensor.

The design and implementation of this technology brought to an international conference paper recently submitted [18], and to the contribution on other works: an international journal article [5], an international conference paper [6], and a technical report [22]. To date, the system is still used to conduct scientific experiments at the IRIDIA lab.

6.1 Future work

An interesting future development consists in the extension of the system design with virtual actuation technology. Virtual actuators are software modules able to modify the augmented reality. A possible way to achieve virtual actuation is to close the communication loop between the robots and the simulator. Following the example of the technology used for virtual sensing, where the virtual sensor readings are delivered by the simulator to the robots, the virtual action

consists in a message delivered by the robots to the simulator. The message encodes the actions that the robots want to perform on the virtual environment. The simulator collects all the virtual actions and modifies the environment accordingly. The modification introduced in the environment by the simulator are immediately perceived by all the robots.

The possible employments of the virtual actuation technology are actuators prototyping and full exploitation of a dynamic virtual environment that changes according to the robots virtual actions. An envisioned application that joins the features of virtual sensing and virtual actuating technologies is the creation of a virtual pheromone for a robotic swarm. Such a component employ both sensing and actuation features, and it is impossible to install on E-Pucks or other low cost robotic platforms. However, its realisation would enable to study stigmergic processes of self organization with real robots experiments. The combination of virtual sensing and actuation technology offers a clean, flexible and software based solution to enable real robots experiments in swarm robotics areas where simulation was the only possible research tool.

Bibliography

[1] Chipmunk website. URL `http://chipmunk-physics.net/`.

[2] Jan Dyre Bjerknes, Wenguo Liu, Alan Ft Winfield, Chris Melhuish, and Coldharbour Lane. Low cost ultrasonic positioning system for mobile robots. *Proceeding of Towards Autonomous Robotic Systems*, pages 107–114, 2007.

[3] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[4] M. Dorigo, M. Birattari, and M. Brambilla. Swarm robotics. *Scholarpedia*, 9(1):1463, 2014.

[5] G Francesca, M Brambilla, A Brutschy, L Garattoni, R Miletitch, G Podevijn, A Reina, T Soleymani, M Salvaro, C Pinciroli, et al. Automode-chocolate: A method for the automatic design of robot swarms that outperforms humans. *Under review*, 2015.

[6] Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Lorenzo Garattoni, Roman Miletitch, Gaetan Podevijn, Andreagiovanni Reina, Touraj Soleymani, Mattia Salvaro, Carlo Pinciroli, Vito Trianni, and Mauro Birattari. An experiment in automatic design of robot swarms: Automode-vanilla, evostick, and human experts. In M. Dorigo et al., editor, *Swarm Intelligence (ANTS 2014)*, volume 8667 of *LNCS*, pages 25–37. Springer, Berlin, Germany, 2014.

[7] S. Garnier, F. Tache, M. Combe, A. Grimal, and G. Theraulaz. Alice in pheromone land: An experimental setup for the study of ant-like robots. *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE.*, pages 37–44, 2007.

[8] IRIDIA. Iridia website, 2014. URL `http://code.ulb.ac.be/iridia.home.php`.

[9] Ali Abdul Khaliq, Maurizio Di Rocco, and Alessandro Saffiotti. Stigmergic algorithms for multiple minimalistic robots on an RFID floor. *Swarm Intelligence*, 8(3):199–225, 2014.

[10] Thomas H. Labella, Marco Dorigo, and Jean-Louis Deneubourg. Division of labor in a group of robots inspired by ants' foraging behavior. *ACM Trans. Auton. Adapt. Syst.*, 1(1):4–25, September 2006. ISSN 1556-4665. doi: 10.1145/1152934.1152936. URL `http://doi.acm.org/10.1145/1152934.1152936`.

[11] Wenguo Liu, Alan F. T. Winfield, Jin Sa, Jie Chen, Lihua Dou, Wenguo Liu, Alan F. T. Winfield, Jin Sa, Jie Chen, and Lihua Dou. Towards energy optimization: Emergent task allocation in a swarm of foraging robots.

[12] Alan G. Millard, James A. Hilder, Jon Timmis, and Alan F.T. Winfield. A low-cost real-time tracking infrastructure for ground-based robot swarms. In M. Dorigo et al., editor, *Proceedings of 9th International Conference on Swarm Intelligence (ANTS)*, volume 8667 of *LNCS*, pages 278–279. Springer International Publishing, 2014.

[13] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stphane Magnenat, Jean christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, pages 59–65, 2009.

[14] MVTech Software GmbH. Halcon library website. URL `http://www.mvtec.com/halcon/`. Last checked on November 2013.

[15] Shervin Nouyan, Alexandre Campo, and Marco Dorigo. Path formation in a robot swarm – self-organized strategies to find your way home, 2008.

[16] Paul J. O'Dowd, Alan F. T. Winfield, and Matthew Studley. The distributed co-evolution of an embodied simulator and controller for swarm robot behaviours. In *IROS*, pages 4995–5000. IEEE, 2011.

[17] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[18] Andreagiovanni Reina, Mattia Salvaro, Gianpiero Francesca, Lorenzo Garattoni, Carlo Pinciroli, Marco Dorigo, and Mauro Birattari. Augmented reality for robots: virtual sensing technology applied to a swarm of e-pucks. *Under review*, 2015.

[19] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987. ISSN 0097-8930. doi: 10.1145/37402.37406. URL `http://doi.acm.org/10.1145/37402.37406`.

[20] Onur Soysal and Erol Sahin. Probabilistic aggregation strategies in swarm robotic systems. In *IEEE Swarm Intelligence Symposium*, pages 325–332, 2005.

[21] William M. Spears, Diana F. Spears, Jerry C. Hamann, and Rodney Heil. Distributed, physics-based control of swarms of vehicles. *Autonomous Robots*, 17:137–162, 2004.

[22] Alessandro Stranieri, Ali Emre Turgut, Gianpiero Francesca, Andreagiovanni Reina, Marco Dorigo, and Mauro Birattari. Iridia's arena tracking system. Technical Report TR/IRIDIA/2013-013, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, November 2013.

[23] Ken Sugawara, Toshiya Kazama, and Toshinori Watanabe. Foraging Behavior of Interacting Robots with Virtual Pheromone. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3074–3079, Los Alamitos, CA, 2004. IEEE Press.

[24] V. Trianni. *Evolutionary Swarm Robotics. Evolving Self-Organising Behaviours in Groups of Autonomous Robots*, volume 108 of *Studies in Computational Intelligence*. Springer Verlag, Berlin, Germany, 2008.