UNIVERSITÉ LIBRE DE BRUXELLES
École Polytechnique de Bruxelles
IRIDIA - Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

On the Design and Implementation of an Accurate, Efficient, and Flexible Simulator for Heterogeneous Swarm Robotics Systems

Carlo PINCIROLI

Promoteur de Thèse:
Prof. **Marco DORIGO**

Co-Promoteur de Thèse:
Prof. **Mauro BIRATTARI**

Thèse présentée en vue de l'obtention du titre de
Docteur en Sciences de l'Ingénieur

Année académique 2013-2014

Abstract

Swarm robotics is a young multidisciplinary research field at the intersection of disciplines such as distributed systems, robotics, artificial intelligence, and complex systems. Considerable research effort has been dedicated to the study of algorithms targeted to specific problems. Nonetheless, implementation and comparison remain difficult due to the lack of shared tools and benchmarks. Among the tools necessary to enable experimentation, the most fundamental is a simulator that offers an adequate level of *accuracy* and *flexibility* to suit the diverse needs of the swarm robotics community. The very nature of swarm robotics, in which systems may comprise large numbers of robots, forces the design to provide *runtimes that increase gracefully with increasing swarm sizes.*

In this thesis, I argue that none of the existing simulators offers satisfactory levels of accuracy, flexibility, and efficiency, due to fundamental limitations of their design. To overcome these limitations, I present ARGoS—a general, multi-robot simulator that currently benchmarks as the fastest in the literature.

In the design of ARGoS, I faced a number of unsolved issues. First, in existing simulators, accuracy is an intrinsic feature of the design. For single-robot applications this choice is reasonable, but for the large number of robots typically involved in a swarm, it results in an unacceptable trade-off between accuracy and efficiency. Second, the prospect of swarm robotics spans diverse potential applications, such as space exploration, ocean restoration, deep-underground mining, and construction of large structures. These applications differ in terms of physics (e.g., motion dynamics) and available communication means. The existing general-purpose simulators are not suitable to simulate such diverse environments accurately and efficiently.

To design ARGoS I introduced novel concepts. First, in ARGoS accuracy is framed as a property of the experimental setup, and is tunable to the requirements of the experiment. To achieve this, I designed the architecture of ARGoS to offer unprecedented levels of modularity. The user can provide customized versions of individual modules, thus assigning computational resources to the relevant aspects. This feature enhances efficiency, since the user can lower the computational cost of

unnecessary aspects of a simulation.

To further decrease runtimes, the architecture of ARGoS exploits the computational resources of modern multi-core systems. In contrast to existing designs with comparable features, ARGoS allows the user to define both the granularity and the scheduling strategy of the parallel tasks, attaining unmatched levels of scalability and efficiency in resource usage.

A further unique feature of ARGoS is the possibility to partition the simulated space in regions managed by dedicated physics engines running in parallel. This feature, besides enhancing parallelism, enables experiments in which multiple regions with different features are simulated. For instance, ARGoS can perform accurate and efficient simulations of scenarios in which amphibian robots act both underwater and on sandy shores.

ARGoS is listed among the major results of the Swarmanoid project.[1] It is currently the official simulator of 4 European projects (ASCENS,[2] H2SWARM,[3] E-SWARM,[4] Swarmix[5]) and is used by 15 universities worldwide. While the core architecture of ARGoS is complete, extensions are continually added by a community of contributors. In particular, ARGoS was the first robot simulator to be integrated with the ns3 network simulator,[6] yielding a software able to simulate both the physics and the network aspects of a swarm. Further extensions under development include support for large-scale modular robots, construction of 3D structures with deformable material, and integration with advanced statistical analysis tools such as MultiVeStA.[7]

[1] http://www.swarmanoid.org/

[2] http://ascens-ist.eu/

[3] http://www.esf.org/activities/eurocores/running-programmes/eurobiosas/collaborative-research-projects-crps/h2swarm.html

[4] http://www.e-swarm.org/

[5] http://www.swarmix.org/

[6] http://www.nsnam.org/

[7] http://code.google.com/p/multivesta/

To my family

Acknowledgements

I would like to express my deep gratitude to Prof. Marco Dorigo for giving me the opportunity to work at IRIDIA, a laboratory with a lively and stimulating environment. Marco's constant support in the development of my work has been a precious source of motivation. During my years at IRIDIA, Marco's advice and criticism have been crucial to shape me as a researcher and as a person. I look up to Marco for his ability to lead, and for his sharp vision of future research directions.

I would also like to thank Dr. Mauro Birattari. The interactions I had with Mauro during my doctoral studies exceeded his role as a supervisor. While teaching me how to be a good researcher, Mauro has also helped me to accept my limits and strengthen my abilities. I am proud to call Mauro a friend.

I also wish to thank Prof. Andrea Roli for the interesting and pleasant discussions we had. I cherish our collaborations, and I consider them a precious occasion to learn and grow as a researcher.

I wish to thank the people who helped me design and develop ARGoS: Frederick Ducatelle, Gianni Di Caro, István Fehérvári, Michael Allwright, and Vito Trianni. Without their feedback, criticism, and contributions, ARGoS probably would not exist.

IRIDIA is a fantastic working environment populated by a variegated gang of remarkable characters. My doctoral period has spanned three 'generations' of *iridians*. While people joined and left, IRIDIA's unique mixture of professionalism, creativity, and fun remained intact. Thanks to Hughes Bersini and Thomas Stützle for contributing to the creation of the IRIDIA gang and for constantly nurturing it.

Among those of the 'old' (pre-Swarmanoid) generation, I wish to thank Anders Lyhne Christensen, Bruno Marchal, Christos Ampatzis, and Elio Tuci for the interesting discussions and the beers we shared.

Among those of the Swarmanoid generation, I first wish to thank Rehan O'Grady. His excellent vision and communication skills have been an invaluable source of inspiration for me.

I would also like to thank Manuele Brambilla, Giovanni Pini, Eliseo Ferrante, Marco Montes de Oca, and Arne Brutschy. You are a pleasure to work with and great friends.

I also wish to thank Alexandre Campo, Alessandro Stranieri, Antal Decugnière, Francesco Sambo, Matteo Borrotti, Navneet Bhalla, Nithin Mathews, Prasanna Balaprakash, and Yara Khaluf for having contributed to making IRIDIA the unique place it is.

I want to thank the 'new' generation of *iridians*, who is taking good care of IRIDIA's traditions of great research and epic fun: Anthony Antoun, Dhananjay Ipparthi, Gaëtan Podevijn, Gabriele Valentini, Gianpiero Francesca, Giovanni Reina, Leonardo Bezerra, Leslie Pérez Cáceres, Lorenzo Garattoni, Roman Miletitch, Tarik Roukny, and Touraj Soleymani.

Special thanks to my officemates: Ali Turgut and Kiyohiko Hattori. It was great fun to chat and share the office with you.

I am grateful to Muriel Decreton, for her kindness and our chats, and to Carlotta Piscopo, for always finding a way to make me smile.

During my doctoral studies, I had the luck to participate to the ASCENS project. This project gave me the opportunity to interact with extraordinary researchers whose interests and backgrounds are different from mine, motivating me to widen my knowledge and follow new paths. I wish to thank in particular Prof. Martin Wirsing for his constant support and encouragement. I would also like to thank Prof. Ugo Montanari, Prof. Franco Zambonelli, Michele Loreti, Rosario Pugliese, Francesco Tiezzi, Nora Koch, Matthias Hölzl, Mariachiara Puviani, and Annabelle Klarl for the stimulating and pleasant discussions we had.

I would also like to thank Prof. Francesco Mondada, Michael Bonani, and their group at EPFL for their essential role in providing the robots we use at IRIDIA, and for keeping them in working condition.

I would like to thank Rachael for having dedicated time and patience to reading and correcting my English, both written and oral, on a daily basis. Your love and encouragement make my life complete.

My final thought is to thank my family, to which I dedicate this work. You gave me the freedom and the means to choose my own path, and supported my choices unconditionally and lovingly.

Contents

List of Figures

Chapter 1

Introduction

Swarm robotics (Beni, 2005) is a scientific field that originated from collective robotics and swarm intelligence (Bonabeau et al., 1999; Dorigo et al., 2000; Dorigo and Birattari, 2007). With respect to collective robotics, swarm robotics shares the fundamental challenges, i.e., designing effective control strategies for large groups of autonomous robots to perform complex tasks. With respect to swarm intelligence, swarm robotics shares principles and methods to achieve coordination (Beni, 2005): fully distributed control, local communication and sensing, and self-organization (Camazine et al., 2003).

The observation of natural swarm systems offers the primary motivations to pursuing this line of research. Natural swarm systems display a number of desirable properties, such as *robustness* to environmental perturbations, individuals mistakes, and/or death of a part of the swarm; *adaptability* to new operational and environmental conditions; and *scalability*, that is, the ability to maintain the overall performance within acceptable bounds for wide ranges of the swarm size (Şahin, 2005).

Since its origin in the 90's, swarm robotics has been linked to research in natural swarm intelligence systems. Researchers have applied models of natural swarm systems to robotic scenarios, either to propose new algorithms based on the original model (Balch et al., 2006; Schmickl, 2011), or to validate the model itself (Balch et al., 2006; Garnier, 2011; Mitri et al., 2013). In the first case, natural swarm systems serve as a basis to understand how an artificial swarm system of comparable characteristics can be realized. Experiments typically assess the performance of the artificial system, and do not consider the degree of similarity of the behavior of the artificial system and its natural counterpart. Conversely, in the second case, swarm robotics systems are used as fully controllable models of their natural counterpart, and experiments are designed to confirm or negate a hypothesis on the natural swarm system under study. As an example of the former case, Garnier et al. (2008) proposed an aggregation algorithm inspired by Jeanson et al. (2005)'s study on cockroaches. Example works for the latter case are Beckers et al. (1994)'s robot implementation of Deneubourg et al. (1990) ant clustering model, and Garnier et al. (2013)'s work on ant route construction.

In the last decade, the research scope has widened considerably. Besides the original *scientific* trend, a new trend has emerged that emphasizes the *engineering* aspects of the design of swarm robotics systems (Brambilla et al., 2013; Dorigo et al., 2014). The engineering trend finds its *raison d'être* in the development of new approaches to tackle problems whose current solution is either impractical, dangerous, or non-existent (Hinchey et al., 2007). A few examples of long-term applications are power-line maintenance, unexploded ordnance removal, search-and-rescue in disaster areas, space exploration, construction, deep-underground mining, underwater environment restoration, and nanosurgery.

Research in swarm robotics is still at an early stage. The path to realize artificial systems that match the performance of natural swarms remains long and presents many open problems. To date, the major achievements in this field consist of algorithms that tackle specific problem instances. The performance of these algorithms strongly depends upon the context in which they are developed (i.e., hardware capabilities and assumptions on the environment). Given this state of affairs, reproducing results and comparing algorithms is difficult, and this hinders the development of the research field as a whole. Despite the importance of this issue, little work has been devoted to a standardization of the robot capabilities, and to the creation of common platforms for experimentation.

In this thesis, I deal with the problem of simulating swarm robotics systems. Simulation is a fundamental tool for experimentation in this field, and its importance often surpasses that of the robots themselves. Over the last decade, a wealth of different robots suitable for swarm robotics has appeared. In addition to the classical two-wheeled robots such as the Jasmine[1], the Alice (Caprari et al., 1998), and the e-puck (Mondada et al., 2006), new concepts for ground robots (e.g., the Kilobot (Rubenstein et al., 2012)), flying robots (e.g., the AR-Drone (Krajník et al., 2011)), and self-assembling robots (e.g., the s-bot (Mondada et al., 2004)) have been introduced to study algorithms for several applications, such as distributed exploration, monitoring, and morphogenesis. However, today's main challenge remains the prohibitive cost of building and maintaining dozens or hundreds of robots in working condition (Cao et al., 1997; Carlson et al., 2004). For this reason, swarm algorithms are more easily prototyped and analyzed in simulation. Simulated ex-

[1] http://www.swarmrobot.org/

periments do not risk harming humans and robots. A simulator offers controllable and repeatable experimental conditions, which are vital to assess the performance of an algorithm. Additionally, simulations can be repeated hundreds of times, thus enabling the fast and inexpensive collection of large amounts of data for analysis. The very nature of swarm systems pushes for experiments involving thousands of individuals, which are simply impossible with current hardware. Simulation can also enable experiments with yet-to-be-built robots, thus providing information *(i)* to support the design of sensors and actuators, and *(ii)* to assess the CPU and memory requirements of the behaviors under study. For all these reasons, in the typical development cycle of swarm algorithms, experimentation with real robots is performed solely as a final step to validate the simulated experiments.

Despite the central role that simulation plays in the design of swarm algorithms, in the current state of the art no significant effort has been devoted to the creation of a genuinely *general-purpose* simulator for swarm robotics. This is in striking contrast with other branches of robotics, such as humanoid, in which practical knowledge of established simulators such as Gazebo (Koenig and Howard, 2004) is considered an asset to obtain a job.

The lack of a general simulator for swarm robotics is due to the fact that creating a dedicated tool is often considered an uninteresting, yet unavoidable, technical chore of the experimental activities. Differently from the case of humanoid robotics, the models involved in a swarm simulation tend to be extremely simple. While the simulation of a humanoid robot requires extensive knowledge of mechanics, the agents involved in a swarm are often represented as collections of interacting mass-less particles. Thus, in the case of humanoid robotics, the effort required to create a personal, single-use simulator is significant; conversely, in the case of swarm robotics, the effort is typically negligible. Consequently, the design challenges posed by a truly general-purpose simulator for swarm robotics are considered too complex and time-consuming to motivate attention.

In this thesis, I argue that the creation of a general-purpose simulator is a necessary step to support the development of swarm robotics. In addition to the mentioned necessity regarding reproducibility and comparability of swarm algorithms, a general-purpose simulator acts as a common ground to enable *cooperation* and *sharing* of code. This aspect is key to create the premises to identify *best practices*—the basic building block of any engineering approach. Moreover, the prospective applications of swarm systems target complex scenarios, thus requiring advanced tools that support the design, implementation, and analysis of complex swarm behaviors. These requirements exceed the capabilities of any single-use simulator, and necessitate innovative, general-purpose platforms.

1.1 Problem Statement

The main focus of this thesis is the design and implementation of a simulator for large-scale, heterogeneous swarm robotics systems. To be useful, a simulator must provide four main features.

The first feature is support for a natural development cycle, to allow the user to prototype solutions, measure their performance, and seamlessly validate them onto real robots. In other words, a simulator must *integrate* a set of tools, thus forming a coordinated software ecosystem that constitutes a productive development framework.

The second required feature is simulating *accurately* the dynamics of the robots and of the environment, as well as the interactions among robots and between the robots and the environment. The simulation must consider the space-network nature of swarm robotics systems: The agents composing these systems possess a body and move in the *physical space*; while, at the same time, the agents communicate, forming a *network*. At any moment, the state of a swarm is the composition of both spatial and network aspects. Spatial aspects include linear/rotational momentum and applied forces; network aspects include topology and exchanged messages. Spatial aspects are inherently continuous, and their dynamics over time are typically modeled by classical mechanics. Network aspects, on the other hand, are discrete and proceed in an event-based fashion.

The third feature necessary for the simulation of a swarm derives from the large-scale and distributed nature of swarm robotics systems. This requires *efficient* simulation techniques. In this thesis, efficiency corresponds to a wise usage of computational resources that aims to minimize the duration of a simulation.

The fourth required feature is *flexibility.* In the context of this work, flexibility refers to the possibility for the user to add new features, such as new robots, sensors, and actuators. A truly flexible simulator can execute any kind of experiment, provided the right set of models.

Accuracy, efficiency, and flexibility are generally viewed as diverging requirements. Reaching a satisfactory degree of accuracy often entails employing models with high computational costs, which negatively affects run-times and, thus, efficiency. Flexibility entails very abstract and general designs, often imposing constraints on the data structures used to store the state of the simulated world. As a result, the benefits derived from a flexible design might limit the type of allowed models (hindering accuracy) and offer little opportunity for optimization (limiting efficiency). A design that offers satisfactory levels of accuracy, efficiency, and flexibility is a challenging problem whose solution requires novel concepts.

1.2 Thesis Structure and Research Contributions

In the past 15 years, development tools for robotics have increased exponentially. In particular, simulators that target fairly general use cases, such as Gazebo (Koenig and Howard, 2004), Webots (Michel, 2004), and USARSim (Balakirsky and Messina, 2006), have been introduced. These simulators, while covering a wide range of requirements for single robot systems, fail to provide the necessary features when increasingly larger-scale multi-robot systems are involved.

This thesis is situated in this gap, providing solutions to realize an accurate, efficient, and flexible simulator for large-scale swarm robotics systems. The result of my work is a multi-robot simulator called *ARGoS* (*Autonomous Robots Go Swarming*). ARGoS was the official robot simulator of the EU-funded project Swarmanoid,[2] and it is currently used in four European projects: ASCENS,[3] H2SWARM,[4] E-SWARM,[5] and Swarmix.[6] ARGoS is open source (under the terms of the MIT license) and is being continually updated.[7]

In the rest of this section, I present an overview of this thesis structure and list the publications I produced during my Ph.D.

In Chapter 2, I frame this work within the related literature. The chapter is divided into two sections. The first provides context, by *(i)* presenting long-term applications of swarm robotics and existing hardware platforms, and *(ii)* sketching approaches to design, modeling, analysis, and implementation. The second section in this chapter is a review of existing simulation approaches and tools.

In Chapter 3, I provide a high-level overview of the design of ARGoS. In Section 3.1, I discuss the main design requirements. In Section 3.2, I describe the structure of the ARGoS architecture. In Section 3.3 I explain how ARGoS organizes spatial data. In Section 3.4, I present one of the most distinctive features of ARGoS: how the simulated space can be partitioned into multiple physics engines executed in parallel. In Section 3.5, I illustrate the parallelization of execution into multiple threads. The work in this chapter was published in:

- C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, T. Stir-

[2] http://www.swarmanoid.org/

[3] http://ascens-ist.eu/

[4] http://www.esf.org/activities/eurocores/running-programmes/eurobiosas/collaborative-research-projects-crps/h2swarm.html

[5] http://www.e-swarm.org/

[6] http://www.swarmix.org/

[7] ARGoS can be downloaded at http://iridia.ulb.ac.be/argos/.

ling, Á. Gutiérrez, L. M. Gambardella, M. Dorigo. **ARGoS: a Modular, Multi-Engine Simulator for Heterogeneous Swarm Robotics**. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 5027–5034. IEEE Computer Society Press, Los Alamitos, CA, 2011.

In Chapter 4, I detail the low-level aspects of the ARGoS implementation. More specifically, I show how the high-level principles and ideas presented in Chapter 3 are realized in practice.

In Chapter 5, I report the experimental activities I conducted to assess the efficiency of ARGoS. In Section 5.1, I introduce the experimental setup. In Section 5.2, I show the results obtained by partitioning the simulated space with multiple instances of the default 2D-dynamics engine. In Section 5.3, I report the results of analogous experiments performed with the default 3D-dynamics engine and 2D-kinematics engine. The work in this chapter was published in:

- C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, M. Dorigo. **ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems**. *Swarm Intelligence*, 6(4):271–295, 2012.

In Chapter 6, I present three complex research experiments that showcase the features of ARGoS, and validate the simulated results through real experiments. In Section 6.1, I present an experiment in which a large swarm of mobile robots performs flocking according to three different strategies. This work shows the validity of the sensor/actuator models provided by ARGoS by default. This work was published in:

- E. Ferrante, A. E. Turgut, C. Huepe, A. Stranieri, C. Pinciroli, M. Dorigo. **Self-Organized Flocking with a Mobile Robot Swarm: a Novel Motion Control Method**. *Adaptive Behavior*, 20(6):460–477, 2012.

In Section 6.2, I illustrate the results of an experiment in which an emergent swarm behavior observed in simulation is confirmed by real-robot experiments. This work was published in:

- F. Ducatelle, G. Di Caro, A. Förster, M. Bonani, M. Dorigo, S. Magnenat, F. Mondada, R. O'Grady, C. Pinciroli, P. Rétornaz, V. Trianni, L. M. Gambardella. **Cooperative Navigation in Robotic Swarms**. *Swarm Intelligence*, available online at `http://link.springer.com/article/10.1007%2Fs11721-013-0089-4#page-1`, 2013.

In Section 6.3.1, I focus on an experiment in which the standard models provided by ARGoS were not able to capture the dynamics of the real robot swarm. I show how the problem was identified, and how ARGoS was extended to deal with the issue. This work was published in:

- G. Pini, A. Brutschy, C. Pinciroli, M. Dorigo, M. Birattari. **Autonomous Task Partitioning in Robot Foraging: an Approach Based on Cost Estimation**. *Adaptive Behavior*, 21(2):118–136, 2013.

In Chapter 7, I report a set of experiments that involve two types of robots—the foot-bot and the eye-bot. In these experiments, the eye-bot coordinate the formation of multiple groups of foot-bots, which must perform several tasks in parallel. This work was published in:

- C. Pinciroli, R. O'Grady, A. L. Christensen, M. Birattari, M. Dorigo. **Parallel Formation of Differently Sized Groups in a Robotic Swarm**. *SICE Journal of the Society of Instrument and Control Engineers*, 52(3):213–226, 2013.
- R. O'Grady, C. Pinciroli, A. L. Christensen, M. Dorigo. **Supervised Group Size Regulation in a Heterogeneous Robotic Swarm**. *9th Conference on Autonomous Robot Systems and Competitions (Robótica 2009).* IPCB, Castelo Branco, Portugal, pages 113-119, 2009.
- C. Pinciroli, R. O'Grady, A. L. Christensen, M. Dorigo. **Self-Organised Recruitment in a Heterogeneous Swarm**. *14th International Conference on Advanced Robotics (ICAR 2009).* Proceedings on CD-ROM, paper ID 176, 8 pages, 2009.

In Chapter 8, I conclude the thesis and discuss directions for future work.

1.3 Other Scientific Contributions

Over the course of my doctorate, I have conducted a number of studies not directly related to the main topic of this thesis. These works touch several topics, and can be divided into two categories: *swarm robotics* and *Boolean network robotics*. For a detailed list, refer to Appendix A.

Chapter 2

Context and State of the Art

The objective of this chapter is to provide a definition of the phrase 'general-purpose simulator'. The content is divided in two parts.

In Section 2.1, I focus on defining the scope of the expression 'general-purpose,' by providing a broad sketch of the most significant research in swarm robotics. The presentation is tailored to highlight those aspects that impact the design of a simulator and its role in the development process of artificial swarm systems.

The meaning of 'general-purpose' is partially linked to the *longevity* of a simulator design. To be successful, a design must overcome the challenge of time accommodating the increasing complexity of future use cases. In Section 2.1.1 I list a number of long-term applications in which swarm-based solutions are envisioned. These applications exceed the current capability of swarm algorithms. As such, they provide vision for the challenges that are yet to be tackled, and establish a reference for the requirements of simulator design.

A truly 'general-purpose' simulator is capable of supporting any kind of robot. In swarm robotics, the number of available robot types has increased considerably over the last ten years, currently comprising a very heterogeneous family of platforms. For the foreseeable future, this trend is set to continue. However, it is possible to identify a number of common capabilities any robot must offer to be suitable for swarm applications. In Section 2.1.2, I briefly illustrate the current state-of-the-art in robot hardware, and encode the capabilities these robots share into a *reference robot architecture*.

Integration is another aspect that contributes to the meaning of 'general-purpose'. A simulator constitutes the center of an ecosystem of software with diverse purposes, namely design, implementation, and modeling.

In Section 2.1.3, I discuss the current approaches in the design of swarm robotics systems, and emphasize the impact of these approaches on the primitive operations a simulator must offer.

In Section 2.1.4, I concentrate on the implementation of swarm algorithms. Enabling this phase of the development process is arguably the most important requirement a simulator must meet. The complexity of swarm systems, however, poses peculiar implementation challenges that fostered the advent of several programming languages based on diverse concepts. In this section, I present the most significant advancements in programming languages for swarm robotics, and discuss the issues for supporting them in a simulator.

The second part of this chapter, Section 2.2, deals with the current state of the art in the simulation of swarm systems.

Section 2.2.1 is dedicated to modeling techniques for swarm systems. Because it is located at the intersection of many disciplines, swarm robotics lends itself to a multitude of modeling approaches. In this section, I argue that despite the wealth of modeling approaches, physics-based simulation remains an unavoidable technique, due to the generality of the approach, and the minimal assumptions upon which it relies.

In Section 2.2.2, I present a number of traditional techniques to model the physics of individual robots, and provide an overview of the most widely used software packages for physics simulation.

Finally, in Section 2.2.3, I discuss the current state of the art in robot simulation. I critically analyze the design of a number of simulators that either enjoy wide acceptance, or provide innovative features that relate to the problem of simulating swarm systems.

2.1 Context: Swarm Robotics Systems

2.1.1 Long-Term Applications

Swarm robotics systems are envisioned for large-scale application scenarios that require reliable, scalable, and autonomous behaviors. In this section, I present a number of such scenarios in which swarm robotics might provide key solutions.

Power-line Maintenance

Power grids are arguably among the fundamental systems upon which most of our technology relies. Maintaining such systems in working condition is critical. Power grids are massively distributed systems scattered across large areas. Currently, maintenance and reparation require substantial human intervention, and these tasks prove dangerous and expensive (Elizondo et al., 2010).

In the last decade, the concept of *smart grid* has started to gain momentum. Smart grids are envisioned as power distribution systems capable of intelligent monitoring, control, and communication, in which human intervention is minimal (Allan, 2012). Solutions based on robotics are essential to the realization of this vision. The large-scale nature of power grids naturally involves solutions that limit centralization and, instead, rely on swarm-based algorithms.

Swarms are envisioned to perform tasks such as monitoring, mapping, fault detection, and reparation (Elizondo et al., 2010; Allan, 2012). For elevated power-lines, UAVs are beginning to be employed for monitoring and mapping activities, while specialized line rovers have performed minor maintenance tasks (Elizondo et al., 2010). Advancements in robotics for underground power-lines have been limited to cable inspection (Allan, 2012).

Unexploded Ordnance Removal

Unexploded ordnance removal is a dangerous but necessary task to improve life conditions in post-conflict zones, and to facilitate the reuse of land for living and cultivation (Habib, 2007).

The current approach to this task involves human specialists, who detect and remove the explosive material manually. While this approach is thorough and accurate, the process is very slow and highly dangerous (Habib, 2007). To mitigate these issues, robots have been introduced to help human operators (Carpenter, 2013). This can be seen as a step towards replacing humans with autonomous robots.

In the near future, robots could be employed to explore large areas, detect landmines, and report on their position. This task is made difficult by the fact that reliably detecting landmines is currently an unsolved challenge, and that navigation on the uneven terrains where landmines are hidden is a slow and dangerous operation for robots (Habib, 2007).

Search-and-Rescue

Search-and-rescue is a scenario in which victims of an accident must be located and brought to safety. Dangers are present both for the rescuers and the victims. The rescuers are exposed to hazardous and often unknown environments, and must perform activities that might endanger their lives as well as those of the victims.

To solve these problems researchers have started to study robotics-based solutions. The most typical incarnation of the search-and-rescue scenario involves a collapsed building whose topology is unknown. The robots must spread, build a map, locate the victims, and bring them to safety.

Countless variants of this scenario have since been introduced. These variants differ in various aspects, such as the type of robots employed (flying, wheeled, insect-like), the type of environment (2D/3D), the nature of the interactions between the robot and the environment (e.g., whether the robots can modify the environment to remove obstacles, or not). An international competition[1] is held every year to showcase and compare research advances.

Space Exploration

Future scenarios of space exploration involve activities such as coordinated observation, planet monitoring, and on-orbit self-assembly (Izzo et al., 2005). The size and complexity of the spacecraft involved in these activities is limited by the cost of building and launching them into space. Additionally, space environments are harsh, and our ability to maneuver a spacecraft promptly to avoid damage decreases with its distance from us (Goldsmith, 1999).

The distributed nature of these scenarios, coupled with the technological limitations in the construction of the spacecraft are compelling space agencies to consider missions in which large groups of small-scale satellites cooperate. To realize this vision, NASA and ESA have launched pioneering programs such as APIES (D'Arrigo and Santandrea, 2006) and ANTS (Curtis et al., 2000) to study methods for distributed autonomous coordination in space.

Collective Construction

The construction of large structures is an activity that currently requires human intervention and involves no automation. Statistics on fatal or serious accidents expose the high risk that construction entails (Li and Poon, 2013).

Autonomous robotics has the potential to offer solutions that could decrease accidents, increase efficiency, and enable construction projects in locations that today would be considered too dangerous for humans.

Research in this field is exploring several multi-robot approaches such as collective block deposition (Petersen et al., 2011), smart materials (Fernandez and Khademhosseini, 2010), and self-assembling structures (Werfel et al., 2006).

Deep-Underground Mining

Deep-underground mining is a problem whose importance is quickly rising, due to the increasing scarcity of near-surface raw materials (Rubio, 2012).

Direct human intervention is generally considered impossible due to the extreme conditions of deep-underground environments—lack of oxygen, absence of light, need for complex supply mechanisms (Donoghue, 2004).

[1] http://www.robocuprescue.org/

Autonomous robotics system have been proposed as a possible solution to perform this task (Green and Plumb, 2011; Huh et al., 2011; Green, 2012). The large environments in which the robots are expected to act require highly parallel and distributed solutions, in which large-scale robot swarms act in a coordinated fashion.

Underwater Environment Exploration and Restoration

Underwater environment exploration (Aro, 2012) and restoration (Rinkevich, 2005) are necessary tasks to cope with the large-scale and potentially catastrophic damages caused by bottom fishing, marine pollution, and climate change (Jackson et al., 2001; Knowlton and Jackson, 2008; C. Hongo, 2012). Restoration is currently performed by human divers, but this activity is both slow and dangerous.

The use of swarms of Autonomous Underwater Vehicles (AUV) is expected to provide viable solutions to improve both the efficiency and the safety of the restoration process.

Nanotechnologies

Nanorobotics is a futuristic branch of robotics that deals with machines operating at scales in the order of 10^{-6} m. The cost-effective production of this kind of robots is an ambitious goal that requires major breakthroughs in molecular manufacturing (Sivasankar and Durairaj, 2012).

Nanorobotics finds potential applications in medicine, to permit treatment of diseases that are currently incurable. Future applications of this technology include cancer treatment, gene therapy, treatment of brain aneurysm, and dentistry.

The limits imposed by the small scale of these robots and the peculiar characteristics of the environment in which they will operate (the human body) create novel and unexplored challenges for robot control (Mavroidis and Ferreira, 2013). The low-resource and simplistic nature of swarm behaviors is likely to play an important role in the development of this technology.

Discussion

These application scenarios can be characterized according to several aspects. For the design of a simulator, the most prominent are scale, heterogeneity, physical properties, and communication means.

Scale. Regarding *scale*, I refer to the fact that the size of the agents involved (more precisely, their sensor and actuation range) is minuscule with respect to the size of the overall scenario. Effective solutions are likely to require systems that involve large numbers of agents acting in parallel and in a coordinated fashion.

Heterogeneity. The complexity of the above application scenarios requires robots with specific abilities. To maintain a high level of performance while keeping the production cost of the swarm low, a viable approach is to employ *heterogeneous* robot swarms, i.e., swarms composed of individuals specialized in a subset of the necessary tasks. At the simplest level, robots could be divided in two classes—*sensing* and *acting*. Sensing robots would be equipped with high-quality sensory devices, and their task would be primarily exploration, map-building, and monitoring. Conversely, acting robots would feature effective devices to modify the environment, e.g., defusing the explosives in unexploded ordnance removal.

Physical Properties. The third distinguishing feature of the aforementioned application scenarios is the *physical properties* of the environment and of the robots. For example, one could consider the way in which robots navigate in the environment. In power-line maintenance, in first approximation, the motion along the power network can be considered one-dimensional. In unexploded ordnance removal, motion is two-dimensional, and can be modeled through kinematics equations. Space exploration is inherently three-dimensional, and in first approximation the agents can be modeled as point-masses. Motion in an underwater environment is similarly three-dimensional, but fluid dynamics plays a fundamental role. In construction tasks, navigating a three-dimensional structure requires mechanical models. Deep-underground mining occurs in muddy and/or heavily littered 3D environments, which can be captured by voxel-based models. Finally, the motion physics of nanosystems is dominated by colloidal effects.

Communication Means. Finally, effective coordination in these scenarios depends dramatically on the available *communication means*. For instance, WiFi is a possible solution in some variants of search-and-rescue scenarios, but is unusable in underwater environments; stigmergy could be employed by the robots to mark already explored areas in unexploded ordnance removal, but it is not available in deep space exploration.

2.1.2 Swarm Robotics Hardware

Several robotics platforms have been used to study swarm robotics applications. However, to date, no platform can be considered a standard in the field, due to heterogeneity of interests and requirements across the research community.

In this section, I identify a number of basic capabilities that a robot must offer to be eligible for swarm robotics applications. These capabilities constitute a robot reference architecture that identifies the type of models and architectures that best suit a simulator for swarm robotics.

Locomotion

The simplest approach to locomotion is a differential drive system that employs two wheels, each powered by an electric gear motor. This design is employed by a large number of robots, such as the Jasmine[2], the s-bot (Mondada et al., 2004), the e-puck (Mondada et al., 2006), the Alice (Caprari et al., 1998), the marXbot (Bonani et al., 2010), and the r-one (McLurkin et al., 2013). The main advantages of wheel-based differential-drive locomotion are *(i)* the simplicity of motion modeling and control, and *(ii)* the fact that motor encoders provide odometry information without the need for additional devices. The major drawback of wheel-based motion is its relatively high cost in terms of components and manufacturing. For this reason, in the design of the Kilobot, Rubenstein et al. (2012) substituted wheels with vibration motors, which offer the same differential-drive motion control at a much lower cost. However, vibration motors do not provide odometry information, and pose strict constraints on the type of surfaces on which the robot can move.

Recent years have seen the introduction of a large number of affordable flying robots. To date, the most successful design approach is the quad-rotor configuration. Quad-rotors are inexpensive to build, and easy to model and control (Mellinger and Kumar, 2011). Notable examples in this category are Parrot's low-cost commercial robot AR.Drone (Krajník et al., 2011), and the open-source designs AeroQuad and ArduCopter.

Communication

The ability to communicate is a fundamental requirement in swarm robotics, because it enables the interactions upon which self-organization is built. The most common communication modality in swarm robotics is *one-to-many*, in the sense that the data originating from a robot reaches multiple robots within a limited range. Frequently, upon receiving data, a robot is able to detect the relative positioning of its origin. This modality goes under the name of *situated communication* (Støy, 2001).

One way to implement one-to-many situated communication is through *vision*, by using robots equipped RGB LEDs and cameras. Vision has been used to convey the internal state of the robots (Christensen et al., 2008), their role in the swarm (Nouyan et al., 2008), and to indicate docking slots in self-assembly (Christensen et al., 2007). Vision-based communication is possible on many robotic platforms, such as the s-bot,

[2] http://www.swarmrobot.org/

the marXbot, and the e-puck.[3] The primary advantage of vision is the simplicity of its implementation, since colored-blob detection algorithms are freely available in the open-source library OpenCV.[4] The fundamental weaknesses of vision are *(i)* the limited number of bits that can be exchanged per control step, *(ii)* the high computational cost of vision algorithms, and *(iii)* the fact that the communication quality is heavily dependent on proper parameter calibration, which in turn depends on environmental light conditions.

The disadvantages of vision can be partially overcome through short-range wireless communication. Gutierrez et al. (2009) proposed an extension board for the e-puck in which data is exchanged through infrared signals. The board is equipped with 12 modules composed of infrared emitters and receivers. The modules can be controlled individually to simultaneously send different messages. Through this board, robots can achieve a maximum bandwidth of 160 b/s for each module. The communication board of the marXbot, designed by Roberts et al. (2009), uses two different methods to exchange data and detect the sender. To transmit data, a radio signal is used; the detection of the sender relies on the exchange of infrared signals. The maximum bandwidth of this board is 800 b/s. The principal flaw of these boards is the added cost per unit they entail. A simpler, low-cost, yet limited solution to this problem has been used in the design of the Kilobot. This robot is equipped with a single infrared transmitter/receiver module pointed downwards at the floor. This system works under the assumption that the floor is reflective, and offers a theoretical maximum bandwidth of 9.6 kb/s.

Sensing

Sensing allows the robot to collect information about the environment in which it operates, and about other robots.

Among the many devices mounted on swarm robots, one ubiquitous device is the *proximity sensor*. This type of sensor provides the robot with spatial data concerning nearby obstacles and robots. In its simplest form, the proximity sensor can be used to avoid obstacles; more complex devices that provide long-range data can permit Simultaneous Localization and Mapping (SLAM).

Proximity sensors can be implemented in several ways. An infrared proximity sensor is a combination of two devices: an infrared light emitter and an infrared light receiver. The sensor works exploiting the fact that the emitted infrared light is reflected back upon hitting an object. The sensor is capable of estimating the distance between the emitter and the object by measuring the difference in voltage between the emitted

[3] Through a third party extension board.

[4] http://opencv.org/

light beam (which is known to the device) and the received one (which is measured). These sensors are cheap and easy to integrate on any robot, but their measures typically display high levels of noise. Many robots are equipped with infrared proximity sensors: the Alice, the e-puck, the s-bot, and the marXbot, to name a few. Ultrasound sensors work on a similar principle, i.e., listening to the echo of an emitted sound wave. Ultrasound sensors have a longer range than infrared proximity sensors, but they are more expensive and their measures depend on the composition of the objects upon which the sound waves bounce. Laser scanners are an option that ensures precise measurements, but their high cost renders them unattractive for swarm applications.

Cameras are another widespread sensing device. Camera sensors are relatively inexpensive to include on a robot, and their versatility justifies the extra cost. As discussed, camera sensors enable color-based communication. In addition, they can be used to detect objects and landmarks, and to reconstruct geometric environmental features.

Robot swarms are often equipped with a diverse set of other sensors. Ground sensors allow robots to detect the features of the ground. Infrared sensors, such as those mounted on the e-puck and the marXbot, allow the robot to detect differences in the color of the ground. RFID cards are also employed to achieve stigmergy[5], by storing data on RFID tags scattered through the environment (Johansson and Saffiotti, 2009). Light sensors are employed to provide the robots with a global gradient, marking in this way the position of an important location in the environment (O'Grady et al., 2005), and smoke sensors are used to estimate the origin of a fire (Marjovi et al., 2009).

Assembly and Manipulation

While motion and communication are ubiquitous features in robotics platforms, the capability to assemble with kin robots or manipulate objects is only present in a minority. However, these capabilities are often mentioned among the most interesting a robotic swarm can offer, because they enable self-assembly (Groß and Dorigo, 2008; Patil et al., 2013), and collective construction (Petersen et al., 2011).

Self-assembling robots can be categorized in three families: lattice-based, chain-based, and mobile robot architectures (Yim et al., 2002).

In lattice-based architecture, the robots connect into regular patterns, such as cubic or hexagonal. Some examples of robots in this category are I-Cube (Ünsal et al., 2001), Proteo (Yim et al., 2001), mole-

[5] Stigmergy is a form of indirect communication among members of a swarm. Typically, stigmergy occurs as a byproduct of the modifications individuals perform on the environment. These modifications affect the behavior of other individuals working on the same area. The concept of stigmergy was introduced by Grassé (1959) to explain the fundamental interactions that occur among ants during nest building.

cube (Mytilinaios et al., 2004), and ATRON (Jørgensen et al., 2004). These robots are equipped with multiple docking devices that form rigid structures.

Chain-based architectures are capable of connecting into chain-like or tree-like structures. Differently from lattice-based architectures, chain-based architectures display articulation, allowing them to change the position and orientation of each module dynamically. Prominent examples of this family of robots are PolyBot (Yim et al., 2000), CONRO (Castano et al., 2002), and M-TRAIN (Murata et al., 2002).

Mobile robot architectures are capable of locomotion and self-assembly. The robots falling into this category are able to form different types of structures. Two prominent examples exist: CEBOT (Fukuda, 1991), a heterogeneous collection of robotic modules with sensing and docking capabilities; and the s-bot (Mondada et al., 2004), the first fully autonomous robot platform capable of locomotion and reconfigurable self-assembly into arbitrary shapes.

Robotic hardware for collective construction is still at a very early stage. The essential concerns that complicate the design of fully functional robots are the manipulation/deposition of construction material, and the navigation of partially assembled structures.

Early attempts to tackle these issues target the construction of 2D structures. Everist et al. (2004) have proposed a construction system using specially-designed blocks, and Werfel et al. (2006) achieved comparable results using struts.

Research on 3D structures includes cooperative assembly by complex robots (Sellner et al., 2006; Stroupe et al., 2006), manipulating and maneuvering over specialized blocks (Terada and Murata, 2008), and strut-climbing robots that become part of the structure they build (Detweiler et al., 2006). The first fully autonomous robotic architecture to achieve complex construction of three-dimensional structures is TERMES (Petersen et al., 2011), which is based on an integrated design of a passive block and a robot capable of manipulating/depositing the block and navigating the assembled structure.

Discussion

Robotic hardware for swarm applications is very diverse. A general-purpose simulator that encompasses this wide variety of hardware architectures must prove extremely flexible.

The main source of heterogeneity is the dynamics of these devices. Motion, as well as assembling and manipulation, are obtained in different ways depending on the hardware platform. The primary shared aspect among all these platforms is the fact that swarms are formed by large numbers of identical robots. Heterogeneity and large quantities of robots are intertwined aspects. A large number of specialized robots is less expensive to build, and easier to study and control.

In addition, it is possible to model these systems in an optimized way with tailored approaches. As it will be discussed in Section 2.2.3, this fact has promoted the creation of domain-specific simulators that combine accuracy and efficiency, at the cost of flexibility.

2.1.3 Swarm Robotics System Design

Designing effective swarm robotics systems is a complex task. This complexity is rooted in the fact that swarm robotics systems are located at the intersection between several domains, with which swarm robotics systems share requirements and open problems. First, swarm robotics systems can be seen as an instance of distributed systems. In this perspective, the design of swarm robotics systems aims to achieve effective parallelism, while dealing with asynchronous dynamics and stochasticity. Second, swarm robotics systems are expected to be autonomous, to adapt to changing environmental conditions, and to display robustness to failure. All these requirements are typical of artificial intelligence. Third, the fundamental role of local interactions among the components of a swarm can be modeled as a complex network. Finally, swarm robotics systems share the typical complications of classical robotics systems (cost, energy efficiency, robustness, ...), with the added challenge of enabling mass production of large numbers of robots.

In all these domains individually, sound design approaches and engineering methodologies are currently under study. As a consequence, a general and universally accepted theory of swarm systems remains undefined (Schweitzer, 2003). Such theory is expected to provide an analytical framework to design and predict the behavior of a swarm robotics system prior to its implementation. In particular, this theory should offer a solution to the problem of translating a swarm-level behavior into individual-level behaviors.

Because of the lack of general-purpose methodologies, the design of swarm robotics systems currently focuses on small-scale, specific application scenarios within a limited, well-defined scope. The result of this activity depends strongly on the experience and ingenuity of the designer, and success stories are difficult to generalize. The approaches to designing swarm behaviors can be categorized along many dimensions. Regarding the design of a simulator, three dimensions are particularly important: *who* realizes the design, *where* the design abstraction is placed, and *how* the behaviors are structured.

Who is the Designer?

With regard to the first dimension, design approaches can be divided into *manual* and *automatic*.

In manual methods, the human designer works on the system *directly*. The designer studies ways to achieve self-organization by factorizing the problem into simpler instances and identifying recurring patterns. Most works that take inspiration from natural swarms fall into this category.

Conversely, in automatic methods, the self-organizing behavior is the result of an algorithm configured by the human designer to search for the solution that best fits the requirements. Thus, the designer works on the system *indirectly*. Methods in the fields of reinforcement learning (Sutton and Barto, 1998) and evolutionary robotics (Nolfi and Floreano, 2004) are typical examples of this category.

Manual and automatic methods have two important differences. First, in manual methods, the experience on the problem domain and the ingenuity of the designer strongly affect the quality of the final result; in automatic methods, the quality of the result depends on the algorithm used—the experience of the designer is required more for configuring the algorithm properly, rather than for the problem domain. Second, manual methods tend to yield solutions whose structure is easier to understand, maintain, and modify with respect to those solutions found with automatic methods. On the other hand, automatic methods are usually better at finding unexpectedly good solutions, which would be erroneously discarded by a human designer.

Where is the Design Abstraction Placed?

With respect to the second dimension, i.e., where the design abstraction is placed, design approaches can be categorized into *top-down* and *bottom-up*.

The top-down abstraction views the swarm as a unique entity. At this level, the designer assumes that the mechanisms for coordination are given, and works directly on swarm-level mechanisms.

The bottom-up abstraction considers the swarm as a collection of separate components. The designer develops mechanisms to achieve cooperation among the components in order to perform a task. In swarm robotics, this means that the level of abstraction is placed at the level of the individual robots.

The bottom-up approach is currently the most widespread and successful abstraction. This is due to the fact that a fundamental prerequisite of the top-down abstraction is currently missing—a widely established formalism that describes the behavior of a large-scale swarm system. Until this issue is solved, effective modeling approaches to predict swarm behaviors and tools/programming languages to implement swarm-level behaviors will not be fully achievable. However, research on implementing the top-down abstraction is active and a number of interesting results exist. These results are presented in Section 2.1.4.

How is the Behavior Structured?

With respect to the third dimension, design approaches can be categorized into *monolithic* approaches, *modular* approaches, and *virtual physics*.

Monolithic behaviors are self-contained and typically single-task-oriented. It is not possible to readily identify simpler components in this type of behaviors. Most swarm behaviors from the field of evolutionary robotics fall into this category, as neural networks, the typical structure targeted in evolutionary robotics works, are difficult to analyze.

Modular systems, on the other hand, are explicitly factorized into intercommunicating components, organized in various ways. The most widespread approach to connect sub-behaviors is to consider them as nodes in a graph, in which arcs encode the condition under which a robot switches from the current sub-behavior to another. In the literature, this approach goes under the name of Finite State Machine (FSM) (Minsky, 1967). The type of condition associated to the arcs further qualifies the type of FSM—for instance, if each arc is associated with a probabilistic condition, it is called a probabilistic FSMs. A hierarchical organization of the behavior, such as the subsumption architecture (Brooks, 1986), has also been proposed in the literature.

In virtual physics, the robots are imagined to be immersed in a virtual potential field. This field is 'virtual' because it is derived by a robot from sensor data. The robots move according to the forces that result from the potential field. This approach was introduced by Khatib (1986), and later refined and extended by Reif and Wang (1999), and Spears et al. (2004).

The main difference between the modular and virtual physics methods is that, in the former, the current swarm state is discrete, because it corresponds to the set of the behaviors of the robots; conversely, in virtual physics methods, the state of the system is continuous, as it corresponds to the state of the virtual potential. Thus, in a modular behavior, a state change corresponds to a behavior *switch*, by one or more robots; whereas, in a virtual physics behavior, a state change corresponds to the (dis)appearance of a 'fold' in the potential field. Furthermore, in modular behaviors, self-organization is the result of the interaction among the current behaviors of each individual. Hence, the design process must consider the possible combinations of individual behaviors *explicitly*. In contrast, the virtual potential metaphor expresses self-organization as the *implicit* process of relaxation of the swarm system towards a state of minimum virtual energy.

For this reason, modular methods are a natural choice to structure behaviors in which it is easy to identify discrete phases and/or events. Classical examples in the literature are foraging (Liu et al., 2007), chaining (Nouyan et al., 2008), and self-assembly (Christensen et al., 2007). Swarm behaviors such as task allocation (Gerkey and Matarić, 2004), pattern formation (Pinciroli et al., 2008) and flocking (Ferrante et al., 2013), characterized by a continuous swarm state, are more suitably tackled by virtual physics methods.

Discussion

A development framework for swarm robotics must support all of the above design methods. The proposed categorization maps directly to requirements on the desired architecture of the framework.

To enable manual design methods, a framework must offer suitable functionality to inspect and analyze swarm behaviors. This functionality can be interactive or non-interactive. Interactive functionality is usually offered by a Graphical User Interface (GUI), and allows for *(i)* inspection of the robot states (e.g., current sensor readings), and *(ii)* modification of the course of an experiment, such as adding/removing/moving an object. Non-interactive functionality includes the possibility to run hundreds of experiments and collect statistics.

To enable automatic methods, the framework must be designed to be integrated within other tools. For instance, in evolutionary robotics, the framework must allow the user to employ a neural network library as behavioral code, and give control of when an experiment must be started to the genetic algorithm.

The bottom-up abstraction requires the development framework to provide primitives to control the devices of the individual robots. The top-down approach, on the other hand, requires functionality such as communication mechanisms that are transparent to the developer.

Regarding behavior structuring, the framework must offer *(i)* the correct primitives to interact with the robots, and *(ii)* the means to integrate the behavior with external tools such as an FSM library.

2.1.4 Swarm Robotics System Implementation

One of the most difficult challenges for the implementation of swarm behaviors for real applications is programming. The faceted nature of swarm robotics systems admits multiple approaches to behavior programming.

Analogously to design approaches, one possible categorization of programming approaches is *bottom-up* and *top-down*. Bottom-up approaches correspond to programming the behavior of each robot individually; top-down approaches allow the developer to program the swarm as a single entity, translating the swarm-level program into instructions for each individual automatically.

Bottom-up Programming Approaches

To date, the bottom-up approach is the most widespread. The lack of general, top-down design methodologies often forces the developer to think at the individual level. Thus, it is natural to translate a bottom-up design into a bottom-up implementation.

Additionally, a large number of programming languages can be used to implement robot behaviors in a bottom-up fashion. Virtually all robots are natively programmed in C and offer a C API, making this language a natural choice. On the downside, C forces the programmer to focus on low-level aspects such as memory management and creation of basic data structures (e.g., linked lists and trees). Thus, many practitioners prefer object-oriented languages such as C++ and (occasionally) Java, for the higher-level abstraction they offer and their large library collection. Following the recent increase in computational power available on robotic platforms, scripting languages have also started to become a viable alternative. Among the many, Lua[6] and Python[7] are the most successful. Lua is a very simple and extensible scripting language designed for low-memory embedded systems. Lua is ideal as a low-resource extension language. Python, on the other hand, is a feature-rich, general-purpose scripting language employed for an extensive variety of applications, including artificial intelligence and data analysis.

The bottom-up approach also includes popular middleware for robotics, such as ROS,[8] OROCOS,[9] and YARP.[10] The purpose of this kind of middleware is twofold: *(i)* providing high-level functions for procedures of general use, such as image analysis, pattern recognition, and mapping; and *(ii)* offering primitives to structure and compose complex behavioral code. Despite its advantages, this type of middleware has found little success in the swarm robotics community, due to the relatively high computational resources it requires, and the lack of primitives for real-time and multi-robot coordination.

[6] http://www.lua.org/

[7] http://www.python.org/

[8] http://www.ros.org/

[9] http://www.orocos.org/

[10] http://wiki.icub.org/yarp/

Top-down Programming Approaches

In the last ten years, the research community has started to study top-down approaches. This effort is motivated by the observation that most of the development time in the bottom-up approach is spent implementing low-level, *ad hoc* communication and coordination procedures. For the success of the aggregate behavior, these procedures must prove robust to failure and performance degradation in presence of a high number of interactions per second. The top-down approach, on the other hand, assumes the presence of dedicated mechanisms of coordination and communication, thus allowing the developer to reason at the swarm level.

The first research efforts towards the top-down approach occurred in the sensor network community. An ample set of abstractions and programming languages has been proposed (Mottola and Picco, 2011), some of which have evolved into languages that target robotics applications. To date, two languages offer the most interesting features: Proto and Meld.

Proto (Beal and Bachrach, 2006) is a language that emphasizes the spatial aspects of a computing network. In other words, Proto is a language that programs 'spatial computers,' that is, a collection of connected computing devices scattered in a physical space. The spatial computer is modeled as a continuous medium in which each point is associated to a value, i.e., a field. The primitive operations of Proto act on this field. There are four types of operations: *(i)* point-wise operations, which are applied to each point in a field; *(ii)* restriction operations, which allow for the selection of a sub-field upon which a certain operation is performed (a form of spatial `if` construct); *(iii)* feedback operations, which encode the notion of evolution over time, allowing the developer to store state; and *(iv)* neighborhood operations, which express the flow of information across the medium. Originally, Proto was designed for non-mobile networks. Bachrach et al. (2010) extended the language to include motion by adding the notion of density at any position in the medium. In this perspective, motion is implemented as a mass flow across the medium.

Meld (Ashley-Rollman et al., 2007, 2009) works from a top-down approach by allowing the developer to specify a high-level, logic description of *what* the swarm as a whole should achieve. The low-level (communication/coordination) mechanisms that reify the high-level goals (i.e., the *how*) are left to the language implementation and are transparent to the developer. For this reason, the syntax of Meld follows the declarative paradigm. The core concepts of the language are *facts* and *rules.* A fact encodes a piece of information that the system considers true at a given time. A special kind of fact is an *action*, that encodes an operation that a robot must perform on the environment. Rules have the role of producing new facts, and are triggered as soon as their associated condition is met. Rules can entail side effects, such as falsifying facts in the knowledge base. The language automatically deletes facts that have become false. A computation in Meld consists of applying the specified rules progressively to produce all true facts, until no further production is possible. A powerful feature of the language is the possibility to specify rules that work on a subset of the entire knowledge base. In Meld parlance, these rules are called *aggregates*. Two types of aggregates exist: *(i)* selection rules, that allow to single out an element in the input fact set (e.g., the fact with the maximum value), and *(ii)* computation rules, that perform a computation over the input fact set (e.g., the sum of all the fact values).

Proto and Meld are two very different solutions for the top-down development approach.

One important point on which these languages differ is in the management of the underlying network topology. The continuous medium abstraction of Proto does not allow for explicit representation of the network topology, nor to refer to individual robots in the swarm. In contrast, in Meld, the topology of the network is represented explicitly, and the language can express rules involving specific robots. In fact, the implementation of Meld imposes the distribution of facts across robots, and, by design, each true fact is stored in a single robot. If a rule requires facts known by different robots, the robot executing the rule requests the fact base of the robot that owns the necessary fact. To make communication possible, every robot `R` stores a fact `neighbor(R,N)` for each robot `N` in its neighborhood. This feature is desirable in applications such as modular robotics and self-assembly, in which the relative positioning of robots often affects the overall behavior.

Another important difference between Meld and Proto is their syntax and semantics. Meld follows the declarative programming paradigm. Meld rules are expressed as logic statements, and computation occurs by iteratively applying rules, producing new facts. Thus, in a non-trivial Meld program, it is difficult to predict the possible effects of a certain rule solely from the program source. Debugging a Meld program requires carefully following the chain of productions. This issue is exacerbated by the fact that facts might be deleted as a result of robot motion, making it difficult to reproduce erroneous executions. In contrast, Proto is a functional language, with a syntax similar to LISP. Proto statements represent the computation as a composition of mathematical functions. For a human developer, Proto statements are readable and the execution of the program is usually predictable from the source code.

A desirable feature in any programming language is compositionality, i.e., the possibility to organize instructions into functions and libraries.

Compositionality allows developers to factorize large problems into simpler ones, and to combine individual solutions intuitively. The LISP-like syntax of Proto is compositional by design. The Meld syntax, in contrast, offer no such feature.

A final important difference between Proto and Meld is the effect of sensor noise and failures on the execution of a program. In Meld, uncertainty deriving from these phenomena is automatically managed by the fact that the derived facts in the knowledge base are deleted at the beginning of each cycle of the control step. Proto, on the other hand, does not explicitly consider noise and failures.

Discussion

Behavior programming is arguably the most important activity a simulator for robotics must engender. As discussed, many languages are available to implement swarm behaviors.

In terms of the interface with a simulator, the fact that a language follows the bottom-up or the top-down approach is negligible. As explained in Section 2.2.3, a simulator is based on a microscopic model of the system, and naturally exposes robot-level control primitives. A bottom-up programming language encodes behaviors using these primitives directly; the infrastructure of a top-down programming language is constructed upon the same primitives.

The choice of the programming language mainly affects the performance of the simulation. Languages such as C and C++ are compiled into the native binary format of the machine on which the simulation is running. Consequently, their performance depends only on the speed of the machine. High-level programming languages, such as Java, Lua, and Python, are compiled into a binary format (called *byte-code*) meant to be executed by a virtual machine. The advantage of this feature is compatibility—the same compiled script can be run, without any modification, on any platform on which the virtual machine is installed. On the downside, executing a program on a virtual machine entails higher memory usage and longer run-times, due to the inability to optimize the binary format for the host machine. The latter issue is usually tackled through the use of just-in-time compilers (Aycock, 2003), typically at the cost of increased memory usage.

Top-down programming languages are essentially scripting languages executed on networked virtual machines. Thus, they share the challenges discussed above. Additionally, the networked nature of these virtual machines forces the simulator to either include a complete model of the virtual machines that also simulates network communication, or to employ the original virtual machines accepting the performance penalty introduced by socket-based communication.

2.2 Simulation of Swarm Robotics Systems

Real-robot experimentation is difficult due to the unreliability of today's robotic hardware (Cao et al., 1997). As the number of robots involved in an experiment grows, maintaining the swarm in working condition becomes prohibitive (Carlson et al., 2004). Thus, to date, the only method to design and test for scalability is to employ modeling techniques, and validate the results with targeted real-robot experiments.

In this section, I present an overview of the techniques to model robot swarms. In Section 2.2.1, I provide a general view of the modeling techniques. In Section 2.2.2, I focus on physics-based approaches.

2.2.1 Modeling Robot Swarms

The literature abounds in modeling approaches that differ widely in purpose and level of abstraction.

Regarding their purpose, modeling techniques can be qualified with respect to their role in the development life cycle. A *non-executable* model typically describes the required properties of the system prior to its implementation. *Executable* models, on the other hand, reproduce the dynamics of the running system. To date, due to the difficulty of predicting swarm behaviors, non-executable models have received little attention in the literature. Two notable exceptions exist: Winfield et al. (2005), who apply temporal logic to model self-organization; and Brambilla et al. (2012), who propose a multi-level approach to swarm design based on formal methods. The vast majority of work in swarm modeling, however, focuses on executable models.

Furthermore, based on the level of abstraction, we can divide models in three families: macroscopic, microscopic, and mesoscopic.

Macroscopic models represent the system from the global point of view, as a single entity. Many modeling techniques in this family are inspired by chemistry or physics (e.g., thermodynamics and synergetics). These models best capture behaviors such as flocking, aggregation, and diffusion, in which individuals can be represented as gas particles. However, these techniques are not suitable for describing behaviors in which individuals are more complex than randomly walking particles. In most complex swarm behaviors, aspects such as the robot body, its capabilities, and the logic of its decision-making play a fundamental role (Hamann and Schmickl, 2012).

Microscopic models lay at the opposite end of the spectrum, as they consider the properties of each robot individually. Agent-based models inspired from biology fall in this category. However, the applicability of these models to swarm robotics is limited because natural and artificial systems differ in two critical aspects. First, a given swarm behavior can be the result of a variety of individual behaviors (Edelstein-Keshet, 2001); second, robots and animals differ widely in terms of capabilities (e.g., sensing, locomotion) and communication means. As a consequence, models that explain natural behaviors might be extremely suboptimal, or even not applicable to swarm robotics applications. An alternative approach to microscopic modeling based on control theory has been proposed by Gazi and Passino (2003). This approach, while allowing one to prove the stability of a swarm behavior, assumes full synchronization, global communication, and absence of uncertainty.

Mesoscopic models place their focus between the two previous categories by including both micro- and macroscopic elements, thus linking macroscopic phenomena to microscopic events. In the works of Martinoli et al. (2004); Lerman et al. (2005); Correll and Martinoli (2006), the swarm is modeled considering the fraction of robots in a certain state. The time evolution of these fractions is expressed through rate equations. While this method is relatively general and mathematically tractable, it neglects the spatial aspects of a swarm. To overcome this issue, Hamann and Wörn (2008) proposed a modeling approach based on Brownian motion.

2.2.2 Physics-Based Robot Modeling

The ultimate goal of a swarm model is to shed light on the relationship between the macroscopic properties of the swarm and the microscopic interactions of the individuals. To achieve this, the model must be expressed at the suitable abstraction level, and allow for analytical investigations. Despite the effort of the research community in the last decade, and the relatively large number of different approaches available, to date, a commonly accepted 'theory' of swarm systems remains unarticulated (Schweitzer, 2003). Moreover, the heterogeneity of the available approaches is an obstacle to the adoption of diverse methods, due to the steep learning curve of each.

To date, physics-based simulation remains the main technique to design, prototype, and test swarm robotics systems. Physics-based models rely on minimal assumptions about the system, which can be summarized as 'the robots possess a body.' Additionally, physics-based modeling is very general, as it targets a necessary feature of swarm systems—motion.

Modeling Motion

There exist several ways to model the motion of a robot (Gazi and Fidan, 2007). In this section, I present four of the most employed models.

Kinematic Model. The simplest possible model to capture the motion of a robot in space is the kinematic model. In this model, the position

p of a robot follows the law

$$\dot{\mathbf{p}} = \mathbf{u},$$

where **u** is the control parameter. A robot modeled according to this law behaves like a mass-less particle capable of changing direction instantaneously. Because of its low computational cost, this model is employed for motion planning and for proof-of-concept navigation experiments. Additionally, this model is frequently used to study approaches to pattern formation and flocking inspired by virtual physics.

Point-Mass Model. The point-mass model is an extension of the kinematic model in which the robot is represented as a geometrical point of mass m. The motion of the robot follows the law:

$$\begin{cases} \dot{\mathbf{p}} & = \mathbf{v} \\ m\dot{\mathbf{v}} & = \mathbf{u} + \mathbf{f}, \end{cases}$$

where **p** is the position of robot, **v** its speed, **u** is the control parameter, and **f** represents the effect of external forces, such as gravity. This model is used for higher-precision simulations in the same situations in which the kinematic model proves useful.

Differential Steering Model. The differential steering model captures the motion dynamics of a two-wheeled robot. The robot state is $[p_x, p_y, \theta]$, where p_x, p_y indicate the position and θ the orientation of the robot. Denoting with u_r and u_l the right and left linear wheel speeds set by the control system, and with b the inter-wheel distance, the model is:

$$\begin{cases} \dot{p}_x & = \dfrac{u_r + u_l}{2} \cos\theta, \\ \dot{p}_y & = \dfrac{u_r + u_l}{2} \sin\theta, \\ \dot{\theta} & = \dfrac{u_r - u_l}{b}. \end{cases}$$

This model is widely used to capture the motion of non-holonomic robots. In addition, a robot can use it to perform dead reckoning from odometry information.

Fully Actuated Model. The most complete model of robot motion is the fully actuated model. This model captures the complete dynamics of a robot composed of connected rigid bodies under the effect of various forces. The model is:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{u},$$

where $\mathbf{q} \in \mathbb{R}^n$ is the robot state (e.g., $\mathbf{q} = [\mathbf{p}, \mathbf{v}]$), **u** is the control parameter, $\mathbf{M}(\mathbf{q})$ is the inertia matrix, and $\mathbf{f}(\mathbf{q}, \dot{\mathbf{q}})$ is a function that accounts for various effects such as: centripetal and Coriolis forces deriving from robot motion, external forces due to gravity and collisions, and random disturbances. This model captures the dynamics of mechanically complex and modular robots, in which multiple bodies are connected by joints. Often, the term $\mathbf{f}(\mathbf{q}, \dot{\mathbf{q}})$ is expressed as a sum of two terms:

$$\mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{f}^k(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{f}^u(\mathbf{q}, \dot{\mathbf{q}}),$$

in which $\mathbf{f}^k(\mathbf{q}, \dot{\mathbf{q}})$ represents known forces, and $\mathbf{f}^u(\mathbf{q}, \dot{\mathbf{q}})$ represents unknown disturbance. The inclusion of the unknown disturbance is employed to develop algorithms that provide robustness to noise and/or perturbations.

Software for Physics Simulation

Several software packages for physics simulation (i.e., *physics engines*) are available. In their simplest form, these packages enable the simulation of the motion of a body in the space. Additionally, many physics engines include functionality such as collision detection and management, and provide means to interact with the internal spatial data structure and implement custom space-aware algorithms. Often, these software packages constitute the core component around which robot simulators are built. In this section, I present the most significant physics engines currently available. All of these physics engines are based on the fully actuated model.

Chipmunk Physics. Chipmunk Physics[11] is a fast 2D physics engine written in C. It is designed to offer high performance for gaming and scientific applications. Chipmunk supports rigid bodies, and spatial data is organized into a space hash (Teschner et al., 2003). For this reason, Chipmunk excels at simulating hundreds of objects of similar size. To maintain high performance, Chipmunk internally uses an Euler integrator. The basic version of Chipmunk is free and open source, and an enhanced version is available for a fee. In addition to the C interface, Chipmunk offers an API in Objective C and can be employed as a plug-in for the Unity Game Engine[12].

Box2D. Box2D[13] is a free and open source physics engine designed to perform fast simulations of rigid bodies. Similar to Chipmunk, Box2D employs an Euler integrator. However, differently from Chipmunk, in Box2D objects are stored into a dynamic AABB tree (Zachmann, 2002). This structure is essentially a binary tree optimized to provide best

[11] http://chipmunk-physics.net/

[12] http://unity3d.com/

[13] http://box2d.org/

performance when the size of the simulated objects is heterogeneous. Box2D is written in C++, and offers an API in Objective C.

Open Dynamics Engine. Open Dynamics Engine (ODE)[14] is a 3D physics engine that targets the simulation of rigid bodies. Over the last decade, ODE has been used for both gaming and scientific applications. In comparison to modern physics engines, the scalability of ODE for increasing numbers of connected objects is known to be poor. ODE offers two integration methods, a fast but unstable Euler integrator, and a slower but more stable fourth-order Runge-Kutta integrator. ODE was originally written in C, but recently a C++ API has been added by the developer community.

Bullet. Bullet[15] is a state-of-the-art 3D physics engine capable of simulating both rigid and soft bodies (cloth, ropes, deformable objects). Internally, spatial data is organized into a dynamic AABB tree (Zachmann, 2002). Bullet can handle a large number of objects through CPU multi-threading and its experimental GPU interface. Bullet offers both Euler and Runge-Kutta integrators. Since its inception, Bullet has enjoyed increasing success and is now employed in a wide variety of software for gaming, visual effects for cinema, and scientific simulations. It is free and open source software, written in C++. Bullet can interoperate with a large number of software packages such as Unity Game Engine[16], Autodesk Maya[16], and Blender[17].

NVIDIA PhysX. NVIDIA PhysX[18] is another state-of-the-art 3D physics engine capable of a wide variety of simulations, including rigid and soft body dynamics, vehicle dynamics, volumetric fluid simulation, and cloth simulation. PhysX can handle a very large number of objects through its native support for multi-threading both on CPU and GPU, and integration with NVIDIA Physics Processing Unit (PPU). PhysX is proprietary software distributed with a free-to-use license for educational purposes. NVIDIA PhysX finds applications in gaming, CAD, visual effects for cinema, and scientific simulations. It offers a C++ interface and it is included in software packages such as Unity Game Engine[8], Autodesk Maya[20], Unreal Engine[19], and Microsoft Robotics Studio[20].

[14] http://ode-wiki.org/wiki/index.php?title=Main_Page

[15] http://www.bulletphysics.org/

[16] http://www.autodesk.com/maya

[17] http://www.blender.org/

[18] https://developer.nvidia.com/physx

[19] Since version 3, http://www.unrealengine.com/

[20] http://www.microsoft.com/robotics/

Vortex. Vortex[21] is a commercial 3D physics engine designed to simulate complex multi-body simulations and robotics manipulation with high fidelity. It includes rigid body dynamics and a particle engine. Vortex is written in C++ and offers a wide variety of plug-ins to simulate sensing, grasping, and vehicle dynamics. Vortex has been employed in a large number of applications in the military sector, as well as in gaming and scientific simulations.

Newton Game Dynamics. Newton Game Dynamics[22] is a 3D physics engine that simulates rigid body dynamics. Newton Game Dynamics sets itself apart from its competitors due to its deterministic collision solver, that yields truly reproducible simulations. On the downside, the performance of this engine is inferior to Bullet's and PhysX'. A multithreaded version of Newton Game Dynamics (for both CPU and GPU computation) is currently under development, and it promises significantly higher performance. Newton Game Dynamics is open source software written in C++. It has been used for gaming and scientific simulations.

Physics Abstraction Layer. Physics Abstraction Layer[23] (PAL) is a unified interface for the most widespread physics engines. Rather than directly providing low-level functionality to perform physics simulations, PAL allows developers to construct physics-based software abstracting the details of the underlying engine. In this way, the underlying physics engine can be changed without affecting the rest of the application. Currently, PAL supports the following physics engines: Box2D, Bullet, Newton Game Dynamics, Open Dynamics Engine, and NVIDIA PhysX.

2.2.3 Simulators for Robotics

In this section, I review a number of important multi-robot simulators and discuss their features with respect to flexibility and scalability. I restrict my focus to mainstream simulators, i.e., those widely used by the research community, and to less known simulators whose designs relate to ARGoS'. For a broader, although dated, discussion of development tools for robotics applications, including but not limited to simulators, I suggest the survey of Kramer and Scheutz (2006).

Simulators Targeting Flexibility

The design of general and flexible simulators is a relatively recent achievement. Before the 2000s, CPU speed and RAM size on an average per-

[21] http://www.vxsim.com/en/software/why_vortex/index.php

[22] http://newtondynamics.com/forum/newton.php

[23] http://pal.sf.net/

sonal computer were insufficient to support extensible designs while ensuring acceptable simulation performance. In the last decade, a few simulators able to support different types of robots were developed.

Gazebo. Gazebo[24] (Koenig and Howard, 2004) is currently one of the most successful simulators in robotics. It is designed as a general tool for a wide spectrum of applications. Gazebo allows users to define virtually any kind of robot or environment through an XML-based *simulation description format*. This description is subsequently translated into low-level configuration for the physics engines and the other modules. At the time of writing, Gazebo offers two physics engines, ODE and Bullet. The architecture of Gazebo is a publish-subscribe system formed by four main processes, each dedicated to a specific task: physics simulation, sensor simulation, graphical visualization, and coordination of the simulation loop. As a multi-process system, simulations in Gazebo proceed in a parallel fashion. The primary benefits of a publish-subscribe architecture lay in its high level of abstraction and generality, which render Gazebo very flexible. The physics process, for instance, exports a message interface that can interoperate with the rest of the system independently of the specific choice of physics engine. Additionally, the message-centric nature of this architecture lends itself to the creation of powerful analysis tools that act as consumers of the information exchanged between the processes. The benefits of Gazebo's architecture come at the expense of efficiency, for two reasons. First, inter-process communication occurs through files and sockets, which are slow media when the number of robots in a simulation increases. Second, the abstract message interface of Gazebo prevents the optimization of the access to the data structures that store the state of the simulated work. Gazebo is open source software written in C++. It is developed by the Open Source Robotics Foundation[25], which also maintains ROS. The publish-subscribe architecture of Gazebo is inspired by ROS and shares part of the code-base. As a result, robot behaviors developed with ROS can be executed in Gazebo seamlessly.

Webots. Webots[26] (Michel, 2004) is a commercial 3D simulator that targets a wide variety of robotics applications. The Webots architecture is designed around the ODE physics engine. The environment and the robots are described in VRML, and are represented internally through a tree structure. It is possible to interface directly to the underlying ODE state, thus retrieving data useful for sensing, actuation, and creation of user extensions. However, this interface is available only for licensed

[24] http://gazebosim.org

[25] http://osrfoundation.org/

[26] http://www.cyberbotics.com/

copies of the software, and its use requires knowledge of both ODE and Webots internals. The scalability of ODE is known to be poor when compared to state-of-the-art physics engines such as Bullet and NVIDIA PhysX. The main limitation of ODE is the lack of multi-threading, which severely limits its performance. For this reason, Webots is not suitable for large-scale simulations involving thousands of robots. Webots is commercial software written in C++. It offers models for a large variety of common robotic platforms. Robot behavior can be written in C, C++, Java, Python, Matlab, and URBI.

USARSim. USARSim[27] (Carpin et al., 2007a) is a 3D simulator designed for generic robotics scenarios. Since its inception, USARSim has enjoyed increasing usage, and it is now the chief simulation framework for the RoboCup Urban Search-and-Rescue League.[1] With respect to other simulators, the community supporting USARSim has devoted significant effort to the validation of the robot models (Carpin et al., 2007b; Balakirsky et al., 2009). USARSim is built on top of Unreal Engine[28], a commercial game engine released by Epic Games[29]. Unreal Engine provides state-of-the-art technology for both physics simulation and realistic rendering. The models for robots, sensors, and other simulated objects, as well as the map of the environment, are specified in the proprietary format of Unreal Engine. At run-time, USARSim interacts with the Unreal Engine through a socket-based interface. This interface allows users to specify control code for the robots. USARSim also provides advanced debugging and monitoring tools implemented as ‘Unreal spectators’. The network-based architecture of USARSim imposes limits on the number of robots that can take part in a simulation. Consequently, USARSim is unfit for large-scale experiments involving thousands of robots. Additionally, the dependency on a commercial game engine renders adopting and extending USARSim problematic. USARSim is open source software written in C++. It supports a wide variety of robots, and offers mechanisms to simulate wireless communication. USARSim allows the user to specify robot behaviors through Gamebot (Kaminka et al., 2002), MOAST (Balakirsky and Messina, 2006; Balakirsky et al., 2008), and Player (Gerkey et al., 2003).

MuroSimF. MuroSimF[30] (Friedmann et al., 2008) is a general-purpose simulator for multi-robot scenarios involving any kind of robot. The main novelty of MuroSimF is the possibility to model any aspect of a robot with adjustable levels of accuracy. To obtain this feature, physics

[27] http://usarsim.sourceforge.net/

[28] http://www.unrealengine.com/

[29] http://www.epicgames.com/

[30] http://www.dribblers.de/research/simulator.en.php?language=en

simulation in MuroSimF is not based on any mainstream physics engine. Instead, simulated objects are structured into tree-like compounds. Each object is assigned a dedicated model for simulation. Objects can also be linked to simulate arms and grippers. Objects can also be associated to a dedicate controller, thus realizing behaviors whose logic is distributed across the subsystems of a robot. Parallelism in MuroSimF is achieved by submitting computation tasks (e.g., the update of an object) to a scheduler, which distributes tasks across a pre-configured thread pool. The architecture of MuroSimF allows the user to assign computational resources to those aspects of the simulation that are most important *for the current experiment.* This feature is important, because it allows the user to avoid wasting computational resources on unimportant aspects of the simulation. As a consequence, the performance of MuroSimF is remarkable—experiments show that MuroSimF can simulate dozens of robots in real-time with a simulation step as short as 1/1000th of a second. The performance of MuroSimF is currently limited by the fact that only part of the computation can be parallelized. In particular, collision detection and handling in MuroSimF are still executed in a single thread. MuroSimF is open source software written in C++.

OpenRave. OpenRave[31] (Diankov, 2010) is a simulator designed to support prototyping of motion planning and manipulation behaviors in industrial applications. The architecture of OpenRave is organized in four components, internally called *layers*: *(i)* the 'core' layer acts as a coordinator of the activities of the other layers; *(ii)* the 'plug-in' layer defines the basic interfaces for the modules executed in a simulation; *(iii)* the 'scripting' layer offers the means to specify the robot behaviors; and *(iv)* the 'robot database' layer provides a set of functions for the analysis of robot behaviors based on metrics collected at run-time. The execution of a simulation is distributed across multiple threads. One thread is dedicated to managing the environment, including physics and sensor calculations. The other threads execute the control logic. This design choice presents both advantages and disadvantages. The main advantage is the clear separation between environment simulation and control logic. As mentioned, the target use case of OpenRave is motion planning and manipulation for complex industrial applications. Thus, the control logic is likely to entail calculations whose computational cost might be comparable with the simulation of the environment. On the downside, in this design the environment is a shared resource among multiple threads. Consequently, to prevent race conditions and ensure exclusive access to each thread, the entire environment must be locked

[31] http://www.openrave.org/

through a mutex. The assumptions upon which OpenRave's design is constructed are not true for swarm robotics. In swarm robotics, the control logic typically has a negligible computational cost with respect to the simulation of the environment. In addition, it is not acceptable for each access to the environment to be exclusive, as this prevents the computation to exploit parallelism efficiently. OpenRave is open source software written in C++. It is integrated with Player, and allows developers to specify robot behaviors in ROS, Python, and Octave/Matlab.

Morse. Morse[32] (Echeverria et al., 2011) is a simulator designed for general robotics scenarios. It is constructed around the Blender game engine[33], an extension of the Blender 3D rendering software that enables the creation of interactive animations with high-quality graphics. To simulate physics, the Blender game engine uses Bullet. The architecture of Morse is multi-process. Morse allows the user to factorize the computation across processes, assigning a subset of the robots to each of them. In this way, a complex simulation can be executed on a computing cluster, accelerating its execution. The processes are organized into a centralized system, in which a server node coordinates the work of the clients. The performance limitations of a multi-process architecture discussed for Gazebo also apply in the case of Morse. Additionally, the simulator is extended through Python scripts, imposing an abstraction layer over C++ that further affects performance. Consequently, this design is not suitable for large-scale multi-robot scenarios. Similarly to its competitors, Morse is open source software written in C++. Additionally to the Python scripting interface, Morse allows the user to specify behaviors through popular middleware for robotics such as ROS and YARP, and exposes a socket-based interface to enable further third-party bindings.

SwarmSimX. SwarmSimX[34] (Lächele et al., 2012) is a 3D simulator designed for real-time robotics applications, such as hardware- and human-in-the-loop scenarios. The main distinctive trait of SwarmSimX is the fact that the simulation loop is synchronized with wall-clock-time to enable a realistic interactive simulation. SwarmSimX is built around NVIDIA PhysX, although the architecture permits the addition of new physics engines. The architecture of SwarmSimX is extremely generic, as it revolves around three high-level concepts: visual representation, physical representation, and behavioral logic. The visual and physical representations are independently specified. Both representations are structured as trees. The visual representation has the purpose of

[32] http://www.openrobots.org/wiki/morse/

[33] http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine

[34] https://svn.kyb.mpg.de/kyb-robotics

rendering a simulated object; the physical representation simulates its dynamics. The behavioral logic is divided into 'drivers' and 'sensors'. Drivers represent the behavior of a simulated object, while sensors provide drivers with information on the simulated environment. A simulated object is represented through the concept of 'artifact', which is a composition of visual/physical representation and behavioral logic. The architecture of SwarmSimX achieves parallelism by assigning a dedicated thread to each artifact. Among the general-purpose simulators, SwarmSimX offers the best performance for multi-robot scenarios. Experiments in (Lächele et al., 2012) show that SwarmSimX is capable of reaching real-time performance in a simulation with hundreds of quad-rotors. SwarmSimX is open source software written in C++. Behaviors can be specified through ROS. In addition, SwarmSimX is integrated with the TeleKyb framework (Lächele et al., 2013) for quad-rotor control.

ReMod3D. ReMod3D[35] (Collins et al., 2013) is a 3D simulator conceived for self-reconfigurable robotics. The main purpose of this software is to support the design of both physical and behavioral aspects of modular robots. The architecture of ReMod3D is composed of a core module that manages a set of working modules. The working modules are capable of connecting to each other in arbitrary topologies that can be modified at run-time. A notable aspect of the design of ReMod3D is the presence of explicit mechanisms for inter-module communication. The flexibility of this design mainly revolves around the use of the generic C++ container `Boost::any`[36]. This container is used to handle modules, and to envelope exchanged messages across modules. The performance of ReMod3D is remarkable. Experiments show that ReMod3D is capable of simulating hundreds of modules in real-time. This performance is primarily due to the GPU acceleration of the NVIDIA PhysX engine employed to simulate the physics of the modules. However, reaching thousands of simulated modules in real time under the same experimental conditions seems problematic with this design, due to the central role of `Boost::any`. In fact, the generality of this container comes at the cost of forcing the system to perform a large number time-consuming type conversions. ReMod3D is open source software written in C++. To the best of my knowledge, the authors do not provide any integration with third-party software.

V-Rep. V-REP[37] (Rohmer et al., 2013) is among the most flexible and feature-rich simulators available. V-REP targets a wide variety of

[35] http://www.isi.edu/robots/remod3d/

[36] http://www.boost.org/doc/libs/1_55_0/doc/html/any.html

[37] http://www.coppeliarobotics.com/

robotics applications, including navigation, motion planning, and manipulation. The architecture of V-REP is complex and very general. Essentially, the architecture is a composition of modules with specific purposes. Modules perform activities such as physics simulation (currently through Bullet, ODE, or Vortex), sensor data retrieval, and robot control. Modules can be freely added by the user. With respect to the process executing the main simulation loop, a module can be executed in three ways: *(i)* as a different process; *(ii)* in a dedicated thread; or *(iii)* on the same machine and thread. The core of the architecture exports a Lua scripting interface, which is the primary mean to implement modules in V-REP. Modules can be implemented also in C, C++, Java, Python, and URBI. The main strength of V-REP is its large collection of extension libraries. V-REP is equipped with libraries for inverse/-forward kinematics, particle simulation, vision, and path planning. The main limitation of V-REP is its high level of abstraction, which, similarly to other designs analyzed in this section, prevents low-level optimization. V-REP is capable of executing multi-robot simulations, but to the best of my knowledge performance evaluation has not been released. V-REP is written in C++ and it has been recently open sourced for academic use. Its general architecture enables integration with any third-party framework for robot control. Among the many, V-REP offers a ROS node that interfaces with the ROS publish/subscribe network.

Simulators Targeting Efficiency

Stage. Stage[38] (Vaughan, 2008) is a simulator designed to execute navigation experiments involving large groups of robots. The architecture of Stage revolves around a custom-made, 2D physics engine that models robots through the kinematic and the differential steering models. The physics engine is capable of detecting collisions and perform basic ray-cast queries. The shape of the simulated objects is represented as a collection of convex polygons, spatially indexed in a grid to improve performance. Multi-threading is implemented by spawning one worker thread per CPU core. A master thread maintains a list of computational tasks to perform and distributes the tasks to the worker threads. Stage's performance is excellent in navigation-based experiments. Experimental evaluation shows that Stage can simulate thousands of robots in real-time. The underlying disadvantage of Stage is the simplicity of its architecture, which renders it impossible to extend the simulator to perform more complex experiments. For instance, the kinematic nature of the models employed in Stage excludes the possibility to simulate self-assembly and foraging experiments. Another limitation of Stage is the lack of noise in sensors and actuators. Stage is open source software

[38] https://github.com/rtv/Stage

written in C++. It is part of the Player project, and behaviors can be written both through Player and ROS.

Roborobo! Roborobo![39] (Bredeche et al., 2013) is a simple 2D simulator conceived for evolutionary experiments involving robot navigation. Similarly to Stage, Roborobo! is built around a custom-made physics engine. However, Roborobo!'s engine is simpler than Stage's. The robots are represented as particles, and motion simulation is limited to the kinematic model. Collision detection exploits a slow pixel-based method, based on SDL's `getPixel32()` function[40]. Roborobo!'s architecture is single-threaded, and this severely limits its performance. Roborobo! is open source software written in C++.

DPRSim and DPRSim2. The Claytronics project (Goldstein et al., 2009) aims to create a new kind of programmable matter formed by millions of sub-millimeter robots. The technological difficulties to face in order to reach this goal force the researchers in this project to prototype their solutions in simulation. However, the massively large-scale nature of the systems under study exceeds the capability of any known robot simulator to offer acceptable performance. For these reasons, Ashley-Rollman et al. (2011) designed DPRSim, a software that simulates large-scale robot ensembles. DPRSim is based on a modified version of ODE, that is capable of multi-threaded computation with a reduced memory footprint with respect to the original implementation. These modifications allowed Ashley-Rollman et al. (2011) to simulate $\sim 10^5$ spherical nanorobots in real time. To further improve performance and reach the objective of $\sim 10^6$ simulated robots in real time, Ashley-Rollman et al. (2011) designed DPRSim2, an improved version of DPRSim executed on a computing cluster. In DPRSim2, the simulated objects are distributed across the computing nodes. The work of these nodes is coordinated by a dedicated master node, which maintains synchronization by issuing a 'tick' signal that triggers an update across the other nodes. To improve performance, the motion of the objects is not simulated; rather, objects are 'magically' transported to their final destination at the due time. The impressive performance of DPRSim and DPRSim2 stems from design choices tailored for a specific use case. As such, the design of these simulators is instructive but not generalizable. DPRSim and DPRSim2 are open source software written in C++.

[39] https://code.google.com/p/roborobo/

[40] http://www.libsdl.org/

2.2.4 Discussion

Marrying flexibility and efficiency while ensuring satisfactory levels of accuracy is a difficult problem.

The simulators designed for flexibility fulfill their purpose through modularity and abstraction. Modularity is an obvious design choice to achieve flexibility, and its combination with high-level abstraction generates designs that can suit a wide set of robotics applications. However, an abstract and modular architecture forces the developers to work far from key low-level aspects, such as data structures and the related algorithms to handle them. In other words, in an abstract and modular architecture, the presence of layers of abstraction imposes static choices exactly where users would want to make modifications.

Conversely, the simulators designed for speed avoid abstraction and modularity, in favor of an approach that targets a limited set of use cases.

In addition to this trade-off between efficiency and flexibility, the architectures of the analyzed simulators are ultimately wrappers around a physics engine. When the number of robots taking part in an experiment rises, the physics engine becomes an increasingly larger factor in the performance of the simulation. Beyond a certain level of performance degradation, these designs offer no solution to restore acceptable levels of performance in physics simulation.

Summary

The design of a general-purpose simulator for swarm robotics is influenced by several factors.

First, swarm robotics is a research field with a wide range of long-term applications. These applications, despite their diversity, share common aspects: *(i)* the robots have a minuscule scale with respect to the size of the environment; *(ii)* the swarms are likely to be heterogeneous in terms of capabilities and assigned roles; both *(iii)* the physical properties of environments and robots, and *(iv)* the communication means vary considerably across applications.

Second, a simulator must be able to capture the capabilities of the robots composing the swarm in terms of locomotion, communication, self-assembling, and sensing. The wealth of long-term applications is reflected by the abundance in robotic platforms targeted to specific scenarios, pushing for general and flexible simulator designs. At the same time, the large numbers involved in swarm systems necessitate efficient and minimal (i.e., robot-specific) models.

Third, a simulator must act as a platform that enables both the design and the implementation of artificial swarm systems. Currently, the most widespread approach is bottom-up, whereby the designer conceives the system as a collection of intercommunicating individuals. Physics-based simulation is a precious tool to make this approach possible, due to the minimal and general assumptions upon which it is founded.

A large number of physics-based simulators exist in the literature. Those simulators that concentrate on flexibility typically offer modular and general architecture capable of simulating any kind of robot. However, such generality and flexibility are obtained at the cost of efficiency, due to the high level of abstraction of the architectural structure and the mechanisms involved. Other simulators concentrate on efficiency, thus enabling experiments with hundreds, thousands, or even millions of robots. These simulators obtain these remarkable results by sacrificing flexibility and accuracy, in favor of simpler and faster robot models.

Chapter 3

The ARGoS Architecture

In this chapter, I present the architecture of ARGoS. In Section 3.1 I discuss the requirements a simulator for swarm robotics must satisfy. Subsequently, I dedicate one section to each design choice: modularity (Section 3.2), customizable indexing (Section 3.3), execution of multiple physics engines (Section 3.4), and multi-threading (Section 3.5).

3.1 Requirements

The main requirement a simulator must meet is *accuracy*. Accuracy is a function of the fidelity of the individual models used in an experiment. It is a measurable property, whose threshold of acceptability depends on the application. For this reason, it is not uncommon to consider the creation of an application-specific simulator as an integral part of the experimental activities. Ideally, a simulator should display perfect accuracy. However, this is an unrealistic expectation, either for practical limitations in the computational aspects of the employed models (e.g., memory limitations, insufficient precision in floating-point operations, etc.), or for the lack of suitable models (e.g., sensor noise). The limitedness of accuracy with respect to the real phenomena under study is often called *reality gap*. In robotics, the reality gap is a serious issue that hinders the transfer of behaviors developed in simulation onto the real platforms (Meeden, 1998).

The second requirement a simulator must satisfy is *efficiency*. Analogously to accuracy, efficiency cannot be considered an inherent feature of a simulator. Rather, it is a measurable property whose acceptance threshold is set by the user considering the available resources (e.g., time, computation power). It is possible to identify two types of efficiency: static efficiency and dynamic efficiency. *Static efficiency* corresponds to the wise use of computational resources (e.g., time, memory, CPU cores) in a specific experimental setup. *Dynamic efficiency* refers to the degradation of performance due to progressive changes in the parameters of

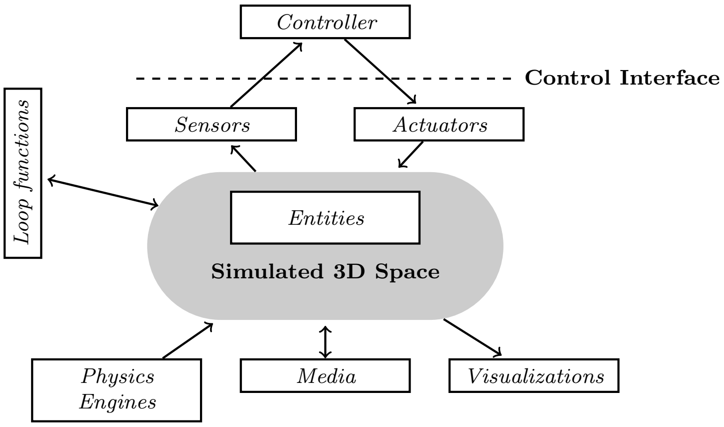


Figure 3.1: The architecture of ARGoS. The white boxes correspond to user-definable plug-ins.

the experimental setup. In swarm robotics, the main parameter of interest in an experimental setup is the swarm size, and dynamic efficiency coincides with the concept of *scalability*.

The third requirement considered in this thesis is *flexibility*. Differently from accuracy and efficiency, flexibility is an inherent property of a simulator design. Flexibility can be generally defined as the capability of a simulator design to encompass a wide set of use cases. In this thesis, a design is considered flexible if it is capable of supporting any kind of robot swarm (see Section 2.1.2) and it allows researchers to perform any kind of experiment of relevance in the field (see Section 2.1.1).

These three requirements are interdependent. Increasing accuracy often forces researchers to employ models with higher computational cost, thus lowering efficiency. A highly flexible design, as discussed in Section 2.2.4, is based on abstractions that prevent the user from optimizing low-level aspects such as data structures and associated algorithms. As a result, efficiency is lowered. In addition, the impossibility to implement algorithms might result in a limitation of the models a simulator can execute, leading to lower accuracy.

3.2 Modularity

In software design, it is common practice to decouple a complex architecture into several interacting modules. As discussed in Section 2.2.3, flexible simulators typically allow the user to modify modules or add new implementations of modules to customize and enhance the functionality of the program. The advantage of modularizing the robot model lies in the possibility to choose which modules to employ for an experiment. Different modules are characterized by different accuracy and computational costs. Thus, the choice of which modules to employ corresponds to the allocation of accuracy where the user deems it necessary. I refer to the possibility to allocate accuracy as *tunable accuracy*.

Tunable accuracy is one of the cornerstones of ARGoS design, as it enhances both flexibility and efficiency. Regarding flexibility the user can define which modules to use for each aspect of the simulation. Efficiency is boosted by the fact that computational resources are allocated only where necessary.

In Figure 3.1, I report a diagram of the ARGoS architecture. The white boxes in the figure correspond to user-definable plug-ins. As illustrated, not only controllers, robot and device models can be selected, but also physics engines and visualizations. In the rest of this section, I describe the features of each plug-in type in depth.

3.2.1 The Simulated 3D Space

The simulated 3D space, depicted at the center of Figure 3.1, is a collection of data structures that contains the complete state of the simulation. This state information includes the position and the orientation of each object such as obstacles or robots. The state of objects composed of different parts or equipped with special devices, such as sets of colored LEDs, is also stored in this space.

The data is organized into basic items referred to as *entities*. ARGoS natively offers several entity types, and the user can customize them or add new ones if necessary. Each type of entity stores information about a specific aspect of the simulation.

For instance, to store the complete state of a wheeled robot, a *composable entity* is used. *Composable entities* are logical containers that are used to group other entities. *Composable entities* can be nested to form trees of arbitrary complexity. The *controllable entity* is a component that stores a reference to the user-defined control code and to the robot's sensors and actuators. The *embodied entity* component stores the position, orientation and 3D bounding box of the robot. The current wheel speed is stored into the *wheeled entity* component. If the robot is equipped with colored LEDs, their state is stored in a component called *LED-equipped entity*.

Entity types are organized in hierarchies. For instance, the *embodied entity* is an extension of the simpler *positional entity*, which contains just the position and orientation of an object, but not its bounding box. These design choices (entity composition and extension) ensure flexibility, enhance code reuse and diminish information redundancy.

Entity types are indexed in efficient data structures optimized for access speed. In this way, the performance of the plug-ins that access the simulated 3D space is enhanced (see Section 3.3).

3.2.2 Sensors and Actuators

Sensors and actuators are plug-ins that access the state of the simulated 3D space. Sensors are granted read-only access to the simulated 3D space, while actuators are allowed to modify it. As explained in Section 3.2.1, information about the simulation state is stored in a number of specialized entities. Sensors and actuators are designed to only access the necessary entities. For instance, the calculations of a distance sensor only need to access information about the *embodied entities* around a robot, and can ignore other entities. In the same way, a robot's LED actuator needs only to update the state of that robot's *LED-equipped* entity.

Tightly linking sensors and actuators to entity components has three benefits: *(i)* these plug-ins can be implemented targeting specific components instead of the complete robot, often resulting in general (rather than robot-specific) models; *(ii)* the robot components associated with sensors and actuators that are not used in an experiment do not need to be updated, avoiding waste of computational resources; *(iii)* new robots can be created faster and more reliably by incorporating existing components, ensuring that all the sensors/actuators depending on them will work without modification. Effects *(i)* and *(iii)* improve flexibility, while effect *(ii)* enhances efficiency.

3.2.3 Physics Engines

As illustrated in Section 3.2.1, an *embodied entity* is a component that stores the position and orientation of a physical object in the 3D space. The state of the *embodied entities* is updated by the physics engines.

As it is explained in Section 3.4, physics engines are assigned non-overlapping portions of the 3D space. At each time step, each physics engine is responsible for the update of the *embodied entities* that occupy its assigned portion of space. This design choice makes it possible to run multiple engines of different types in parallel during an experiment.

Physics engines operate on a custom representation of their assigned portion of the 3D space. For instance, the position (x, y, z) of an object in the 3D space could be stored as (x', y') in a 2D engine. At each time step, the 2D physics engine performs calculations to update its internal representation (x', y') and then transforms it into the common 3D space representation. This design choice enables one to optimize each physics engine's internal representation of space for speed, memory usage and/or accuracy. Currently, ARGoS is equipped with four kinds of physics engines, designed to accommodate the most general use cases: *(i)* a 3D-dynamics engine based on ODE, *(ii)* a 3D particle engine, *(iii)* a 2D-dynamics engine based on the open source physics engine library

Chipmunk,[1] and *(iv)* a 2D-kinematics engine.

3.2.4 Media

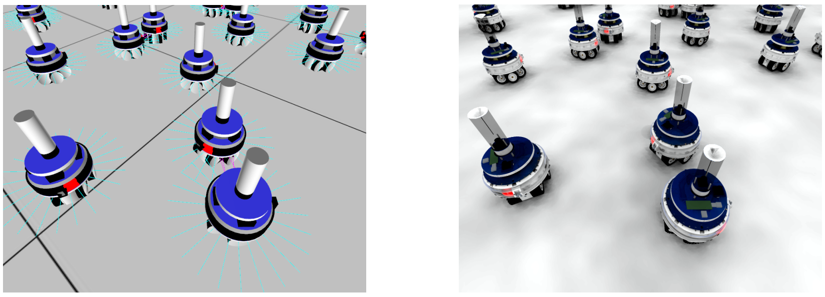
Communication is a fundamental aspect in the simulation of swarm systems. It is only through communication that a swarm can achieve coordination. In Section 2.1.2, I have listed the many forms communication can take within a swarm. Essentially, communication occurs as a spatially localized phenomenon, in that robots can exchange messages only within a limited range. In addition, devices such as cameras and the range-and-bearing system introduce the requirement that information exchange can occur only if the involved robots are in direct line-of-sight. In other words, communication in swarms does not only involve data exchange, but it also includes spatial aspects such as positioning and occlusions.

For this reason, simulating localized communication efficiently is difficult. To allow third-party extension developers to select the best option for each type of communication mean, ARGoS offers two main methods to simulate communication.

The first method simply consists in realizing pairs of dedicated sensors/actuators associated with communication-specific entity types. Every time a robot emits data, the actuator sets the data in the associated communication entity, and the sensors of other robots receive these data by scanning the space for nearby communication entities. While this method is simple to implement, it is often inefficient. The main causes of inefficiency are two. First, if robot *R1* can communicate with robot *R2*, it is often the case that the inverse is true as well — *R2* can communicate with *R1*. This is the case, for instance, of the range-and-bearing system. The second cause of inefficiency is the low locality of the data structures involved in the simulation of communication (see also Section 3.3). In fact, every time a robot scans its neighborhood for communication entities, the data structures that store the wanted entities must be loaded in cache, used once, and discarded to make room for the next scan. Optimizing memory access in this context is difficult, and it is likely to entail a significant increase in the architecture complexity.

To cope with these issues, the ARGoS architecture offers a specialized module type called *medium*. A medium can be considered as a 'physics engine' for communication. A medium stores the information about communicating entities in optimized data structures, and, analogously to a physics engine, upon update it executes the necessary data transfer among communication-specific entities. Media are very general module types, and they can be used to simulate any kind of communication exchange. Currently, ARGoS offers two types of media: *(i)* the range-

[1] http://code.google.com/p/chipmunk-physics/



(a)

(b)

Figure 3.2: Screen-shots from different visualizations. (a) Qt-OpenGL; (b) POV-Ray.

and-bearing medium, and *(ii)* the LED medium. At the time of writing, further types of media are under development: the RFID-tag medium, to simulate RFID-based communication, and the pheromone medium, to enable simulations of insect-like stigmergic behaviors.

3.2.5 Visualizations

Visualizations are plug-ins that read the state of the simulated 3D space and output a representation of it. Three types of visualization are currently available in ARGoS: *(i)* an interactive graphical user interface based on Qt4[2] and OpenGL,[3] *(ii)* a high-quality rendering engine based on the ray-tracing software POV-Ray,[4] and *(iii)* a text-based visualization designed for interaction with data analysis programs such as Matlab.[5]

3.2.6 Controllers

Robot controllers are plug-ins that contain the control logic of the robot behavior for an experiment. An important requirement in the design of a simulator is the possibility to develop code in simulation and then transfer it to the real robots without modification. To meet this requirement, ARGoS provides an abstract *control interface* that controllers must use to access sensors and actuators. The same control interface is also implemented on the real robots. In this way, the user code developed in simulation can be transferred to the real robots without modifications.

Currently, robot controllers are written in C++. In swarm robotics, robots are typically systems with low-end processors such as ARM.[6]

[2] http://qt.nokia.com/

[3] http://www.opengl.org/

[4] http://www.povray.org/

[5] http://www.mathworks.com/products/matlab/

[6] http://www.arm.com/

Thus, transferring code from a personal computer to a real robot requires recompilation. Aside from the recompilation step, however, simulated code is directly usable on real robots.

I intend to integrate other programming languages that would not require recompilation to transfer code from simulation to real platforms. At the moment of writing, the ASEBA (Magnenat et al., 2010) and Lua scripting languages have already been integrated in ARGoS, and further language bindings (e.g., Python, PROTO) are under study.

3.2.7 Beyond Modularity: Loop Functions

It is very difficult to identify a set of features that can cover all the possible use cases of multi-robot systems. Even though some features, such as robot motion, are almost always necessary, many other features depend on the type of experiment considered. For instance, the metrics against which statistics must be calculated depend on the experiment. Also, if the environment presents custom dynamics, such as objects being added or removed as a result of the actions of the robots, these mechanisms need to be implemented in the simulator. The need for specific and often divergent features renders the design of a generic simulator extremely complex. Furthermore, the approach of trying to add a myriad of features in the attempt to cover every possible use case usually renders the learning curve of a tool much steeper, hindering usability and maintainability.

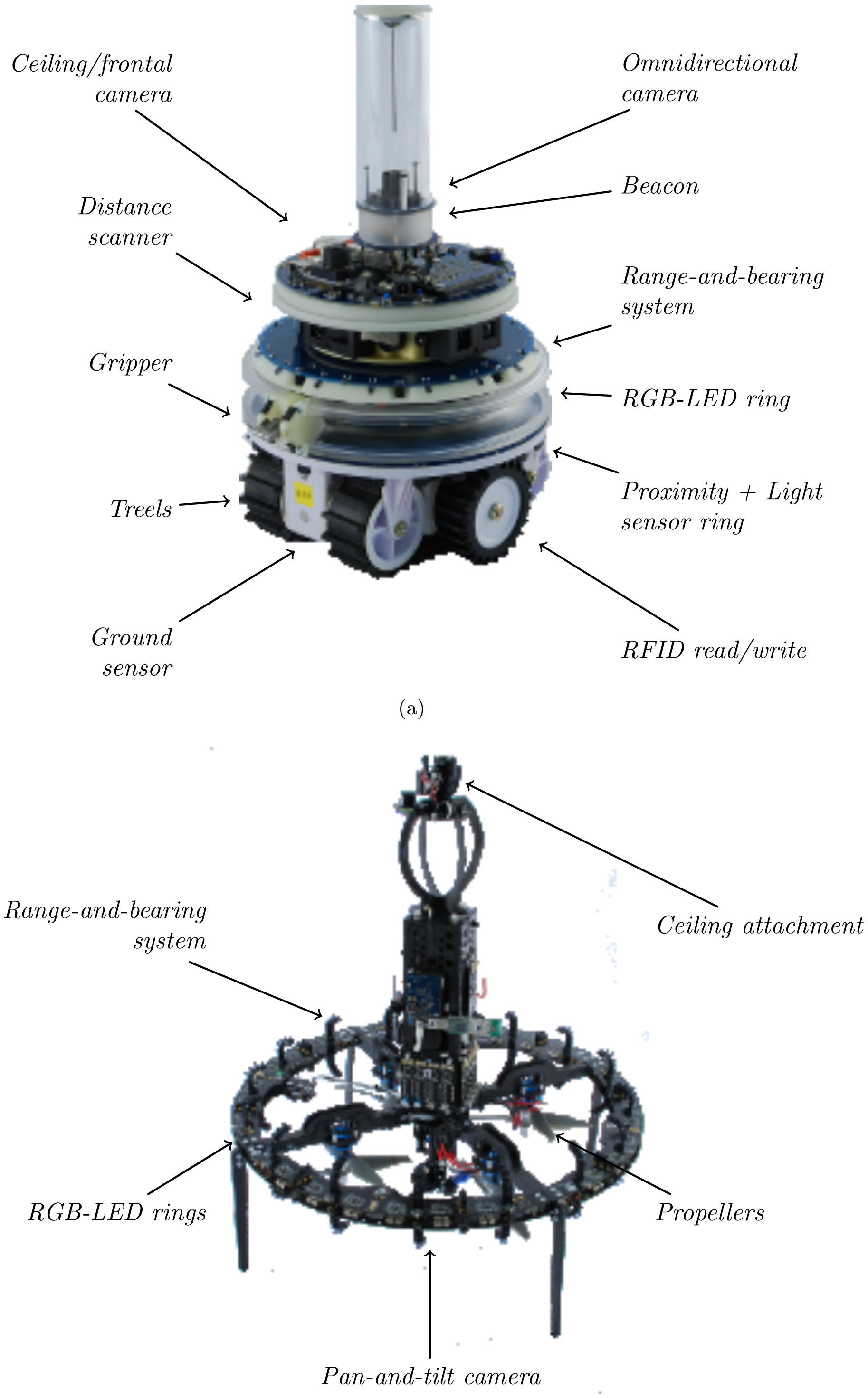
To cope with these issues, I followed the common approach of providing user-defined function hooks in strategic points of the simulation loop. In ARGoS, these hooks are called *loop functions*. The user can customize the initialization and the end of an experiment, and add custom functionality executed before and/or after each simulation step. It is also possible to define custom end conditions for an experiment.

Loop functions allow one to access and modify the entire simulation. In this way, the user can collect figures and statistics, and store complex data for later analysis. It is also possible to interact with the simulation by moving, adding or removing entities in the environment, or by changing their internal state.

Finally, loop functions can be used to prototype new features before they are promoted to the core ARGoS code.

3.2.8 Case Study: The Foot-bot and the Eye-bot

To discuss ARGoS modularity, I describe how two extremely different robots are modeled in ARGoS. The two robots I chose for this case study are the foot-bot (Bonani et al., 2010) and the eye-bot (Roberts et al., 2007). The foot-bot is a ground-based robot that moves with a



(a)

(b)

Figure 3.3: (a) The foot-bot; (b) The eye-bot.

combination of wheels and tracks (called *treels*). The eye-bot is a quadrotor aerial robot. Both robots are equipped with various sensors and actuators that allow them to interact with the surrounding environment. The robots and their devices are depicted in Figure 3.3.

Sensors and Actuators. Both robots are equipped with numerous sensors and actuators, and a few of them present similar features. In the following discussion, I focus on the sensors and the actuators of the two robots that share common aspects. It is important to note that some robot devices may function both as a sensor and an actuator. In these cases, in ARGoS two plug-ins are necessary—one for the sensor and one for the actuator. One of the simplest devices present on both robots is the on-board clock. The state of the clock is read by the clock sensor. Given its simplicity, ARGoS natively offers a single implementation of the clock sensor suitable for all the robots. Both the foot-bot and the eye-bot are equipped with LEDs. However, the specifics of the control interface of the LEDs are different, because the LEDs are distributed differently on the two robots. On the foot-bot, there are two kinds of LEDs: a ring that surrounds the body, composed of 12 RGB LEDs, and a beacon positioned at the center of the robot body. The eye-bot is equipped with two rings of 16 LEDs each: an upper ring and a lower ring. The upper ring is primarily visible from the side, while the lower ring is visible from beneath the robot. Although the control interface for the LEDs is different, the plug-ins share the same code base, thanks to the *LED-equipped entity*. Both the foot-bot and the eye-bot are equipped with cameras. The foot-bot has two: an omnidirectional camera and a camera that can be mounted looking upwards or frontally. The eye-bot has a pan-and-tilt camera, that is, a camera mounted on a rotating device. The implementation of the cameras is specific to each robot. The cameras of the foot-bots are pure sensors, while the eye-bot's pan-and-tilt camera has an associated actuator that controls the attitude of the camera. Notwithstanding these differences, the implementations of the camera sensors are based on a common definition that is extended to suit the specific needs of each camera type, thus ensuring code reuse.

The foot-bot and the eye-bot can communicate with each other through the range-and-bearing communication device (Roberts et al., 2009), which allows the robots in line-of-sight to exchange messages within a limited range. The particularity of this communication device is that a robot, upon receipt of a message, can calculate the relative position (distance and angle) of the sender. The implementation of this device is divided into two parts: the range-and-bearing sensor and the range-and-bearing actuator. The role of the former is to manage the receipt of messages from other robots. The role of the latter is to set the message to send. The implementation of this device is shared between the foot-bot and the eye-bot.

Composing Entities. Both the foot-bot and the eye-bot are implemented as *composable entities*. They share most components. Both are composed by an *embodied entity*, a *controllable entity*, and an *LED-equipped entity*. In addition, the state of the distance scanner is stored in a *distance-scanner-equipped entity*, while the state of the range-and-bearing communication system is stored into a *range-and-bearing equipped entity*. Both the *LED-equipped entity* and the *range-and-bearing equipped entity* are associated to their respective media, enabling communication among the robots. The cameras of the robots are represented by dedicated camera-equipped entities. In particular, the foot-bot contains an *omnidirectional-camera-equipped entity*, while the eye-bot contains a *pan-and-tilt-camera-equipped entity*. The foot-bot possesses a number of entities that are not in common with the eye-bot, such as the *ground-sensor-equipped entity*, the *light-sensor-equipped entity*, the *proximity-sensor-equipped entity*, and the *gripper-equipped entity*. Data on motion actuation is stored into a *wheeled entity* for the foot-bot, and into a *propeller-equipped entity* for the eye-bot.

3.3 Entity Indexing

An important component of any physics-based robot simulation is how spatial data is handled. Spatial data structures are employed for all the most computationally intensive chores, such as collision detection, sensing, and communication. The choice of data structure heavily impacts the performance of a simulator.

The main operations that must be performed over the spatial data structure are *insertion*, *deletion*, *update*, and *query*. *Insertion* and *deletion* typically occur during the initialization and destruction phases of a simulation, and affect performance only marginally. *Update* is usually executed once per loop. *Query* is the most frequent operation, being the basis of the vast majority of the calculations for sensing and collision detection.

For this reason, in the choice of a data structure, priority is usually given to the time complexity of the *query* operation. However, in all but the most trivial cases, the time performance of a query depends on the memory storage characteristics of the data structure. The memory of a modern computer is organized into a multi-level hierarchy. The top-level memory, called L1 cache, is physically connected to the CPU and ensures the fastest access performance. This type of memory is expensive and limited in size. Ideally, the necessary data to perform a computation should not exceed the size of the L1 cache, and all the necessary data should be present in this cache when needed. In practice, however, maintaining the contents of this cache in optimal conditions is a difficult problem. Whenever data is missing, (i.e., a *cache miss* occurs), the processor must wait for the data to be loaded from the lower-level cache, L2, losing precious time. Cache misses can occur at any level of the memory hierarchy, generating a cascade of load requests that eventually reach the main RAM storage. The time taken by a load request increases with the depth of such request in the hierarchy. Consequently, the time performance of a *query* operation on a spatial data structure depends on the way the data is organized in memory. In this respect, an important desired property for a spatial data structure is *locality*, i.e., the capability to pack data in such a way to minimize the number of cache misses.

The spatial data of a physics-based simulation is of two types: point-based data and object-based data (Samet, 2006). Point-based data consists of a vector $\mathbf{x} \in \mathbb{R}^n$, with $n = 2$ for two-dimensional simulations and $n = 3$ of three-dimensional ones. This kind of data typically represents the position of an object in the space. For simulations based on the kinematic or point-mass models, point-based data is a common representation. Object-based data, on the other hand, is represented by an area or a volume in space. For instance, in collision detection, it is common to query the space to discover whether the body of an object is intersecting another body. A common solution to this problem is wrapping every body of arbitrary shape with an axis-aligned bounding box, and calculate intersections among bounding boxes. For each detected bounding box intersection, the bodies therein are checked for intersection.

In ARGoS, spatial indexing is implemented as a *policy* (Alexandrescu, 2001), and it can be used by any plug-in to index entities. The ability to specify the spatial indexing as part of the simulation setup is a unique feature of ARGoS with respect to existing designs. This feature allows the user to fine-tune the performance of the simulation, thus increasing efficiency without affecting accuracy.

3.4 Multiple Physics Engines

The most distinctive feature of ARGoS is the possibility to partition the simulated 3D space into non-overlapping portions of arbitrary size, and assign each portion to a different physics engine. For instance, in an environment formed by several rooms connected by corridors, the different rooms and the corridor could each be assigned a dedicated physics engine. The volumes assigned to 3D engines are specified as right prisms whose size is user-defined, while the areas assigned to 2D engines are specified as polygons in the 3D space. The user must define the effect of a robot crossing the face of a prism (or the side of a polygon). There are two kinds of faces (sides): *walls* and *gates*. Walls cannot be traversed—when a robot tries to cross a wall, the physics engine does not allow the action. Conversely, when a robot traverses a gate, the robot is transferred to another physics engine defined by the user in the experiment configuration file. This migration is performed automatically by ARGoS as robots navigate in the environment.

In practice, this partitioning is obtained by dividing the set of *embodied entities* in multiple subsets, and assigning each subset to a different physics engine. To avoid conflicts of responsibility among the physics engines, I distinguish between *mobile* and *non-mobile embodied entities*. Mobile *embodied entities* are such that their state (position, orientation, 3D bounding box) changes over time. Robots, as well as passive objects that can be pulled or pushed, fall into this category. Non-mobile *embodied entities*, on the other hand, never change state. Typical examples of this category are the structural elements of the experimental arena, such as walls and columns. To ensure the correctness of the state of the simulated space, I impose the condition that mobile *embodied entities* can be managed by only one physics engine at a time. Non-mobile entities, instead, can be associated to as many engines as necessary. In this way, the structural elements of the arena are shared among all the engines, resulting in a consistent representation of the simulated 3D space.

When two robots are managed by different physics engines, they can not interact physically (i.e., collide or grip each other). However, communication and sensing work across the physics engines. For example, ray casting is a typical method to simulate point-to-point communication, occlusion checking and sensing. To check for ray intersections, a sensor queries the simulated 3D space. The simulated 3D space constructs a list of candidate *embodied entities* whose 3D bounding boxes intersect the ray. This step is very efficient because *embodied entities* are indexed in a space index (see Section 3.3). Next, each candidate *embodied entity* queries the physics engine in charge of its update to perform the actual ray-body intersection. The result is then returned to the sensor that issued the check. Thus, even if the check is performed by a physics engine, the sensor does not need to interact directly with the physics engine. This renders sensor development easy, because there is no dependency between sensors and physics engines.

Because robots updated by different physics engines do not interact physically, compenetrations are possible at the border between two engines. It is the user's task to partition the space in such a way as to avoid this phenomenon or to limit its impact on accuracy. A typical method, used, for example, in the experiments of Chapter 5, is to exploit the fact that robots can sense each other across physics engines and implement efficient obstacle avoidance strategies. Another solution is partitioning the space appropriately. For example, in an experiment with flying and

```
Initialize
Visualize the simulated 3D space
while experiment is not finished do
    for all robots do
        Update sensor readings
        Execute control step
    end for
    for all robots do                   } sense+control
        Update robot status
    end for
    for all physics engines do          } act
        Update physics
    end for                             } physics
    Visualize the simulated 3D space
end while
Cleanup
```

Figure 3.4: Simplified pseudo-code of the main simulation loop of ARGoS. Each 'for all' loop corresponds to a phase of the main simulation loop. Each phase is parallelized as shown in Figure 3.5.

wheeled robots, flying robots could be assigned to one physics engine and wheeled robots to another. As long as the flying robots stay sufficiently high, collisions with wheeled robots are not possible.

In Chapter 5 I empirically demonstrate that the use of multiple physics engines results in a significant decrease in simulation time.

3.5 Multiple Threads

To maximize simulation speed, the ARGoS architecture is designed to exploit modern CPU architectures. In practice, this is obtained by parallelizing the main simulation loop reported in Figure 3.4. During the execution of the loop, sensors and visualizations read simulated 3D space, while actuators and physics engines modify it. Thus, simulated 3D space is a shared resource. In multi-threaded applications, simultaneous read/write access on shared resources requires careful design, as access conflicts (*race conditions*) could potentially occur. A typical solution to prevent race conditions is to employ mutexes or semaphores. However, these solutions typically entail significant performance costs (Tanenbaum, 2001). Therefore, to improve performance, I designed the main loop and the simulated 3D space in such a way as to avoid race conditions altogether. These design choices benefit not only performance: the lack of race conditions means that plug-ins do not need to synchronize or manage resource access explicitly, thus simplifying their development.

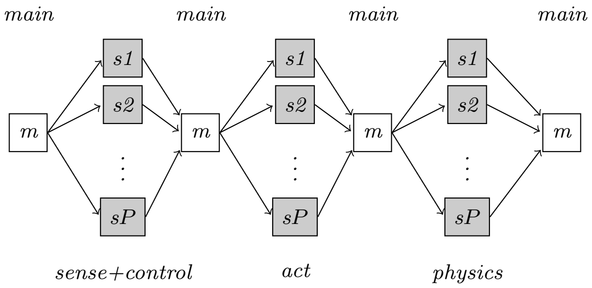
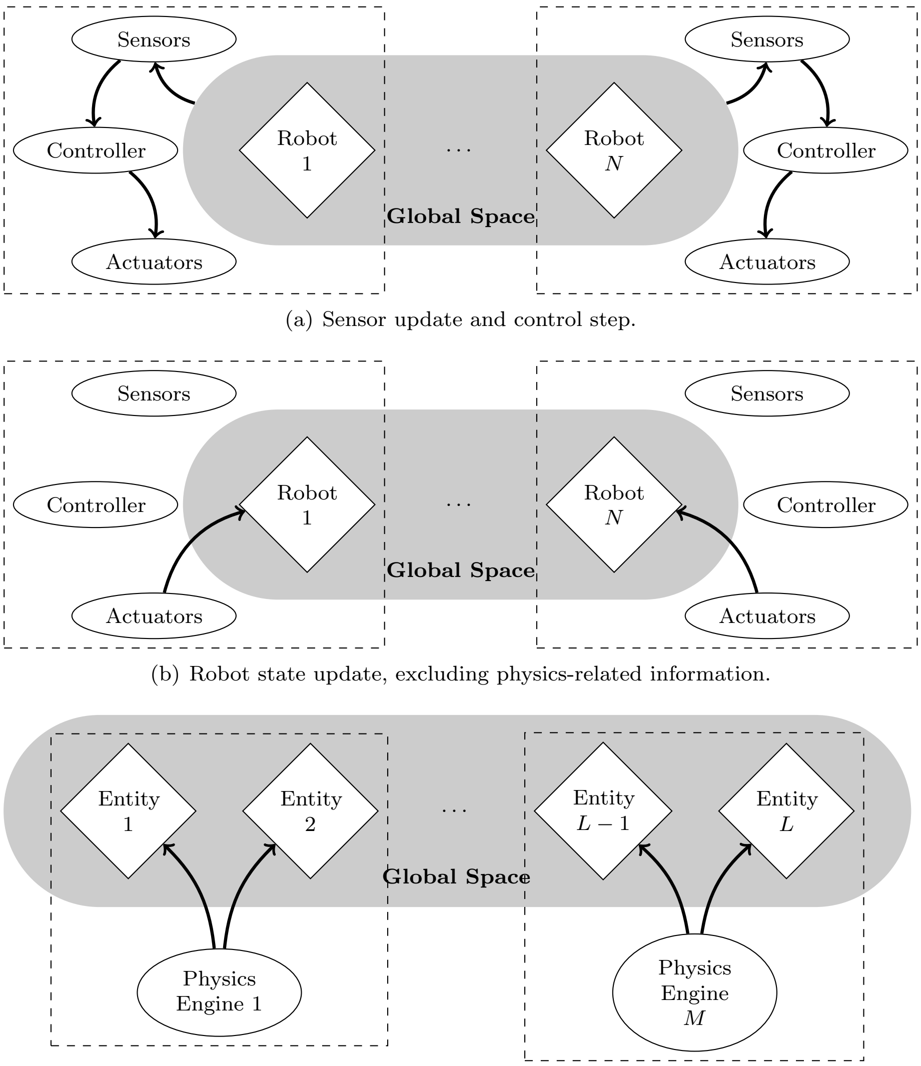


Figure 3.5: The multi-threading schema of ARGoS is scatter-gather. The master thread (denoted by 'm') coordinates the activity of the slave threads (denoted by 's'). The *sense+control*, *act* and *physics* phases are performed by P parallel threads. P is defined by the user.

The main simulation loop is composed of three phases executed in sequence: *sense+control*, *act* and *physics*. In the first phase (*sense+control*, lines 4–7 of the algorithm in Figure 3.4), the robot sensors read the state of the simulated 3D space. The robot controllers use such information to execute the control step. At the end of this phase, the actions chosen by the robot controllers are stored into the actuators, but the actions have not yet been executed. Therefore, this phase is read-only, and race conditions are not possible. In the second phase, *act* (lines 8–10 of the algorithm in Figure 3.4), the actions stored in the actuators are executed. All the component entities of each robot in the simulated 3D space are updated, with the exception of the *embodied entities*. Race conditions cannot occur because, as explained in Section 3.2, each actuator is linked to a single robot component, and two actuators cannot be linked to the same component of a specific robot. In the third and final phase, *physics* (lines 11–13 of the algorithm in Figure 3.4), the physics engines update the state of the *embodied entities*. Race conditions are not possible because, as explained in Section 3.4, in each simulation step, mobile *embodied entities* are associated to one and only one physics engine, and *embodied entities* in different physics engines do not interact. Furthermore, physics engines do not need synchronization, because the duration of the simulation step is set as a parameter in the experiment configuration file and is common to all the physics engines.

Each phase is executed by a set of P slave threads. This parameter is chosen by the user in the experiment configuration file. Experimental evaluation demonstrates that maximum performance is reached when P matches the number of available processor cores (see Chapter 5). A master thread coordinates the beginning and the end of each phase. In Figure 3.5 the master thread is indicated by 'm' and the slave threads are indicated by 's'. Execution proceeds in a scatter-gather fashion. At the beginning of each phase, the slave threads are idle. When the



(a) Sensor update and control step.

(b) Robot state update, excluding physics-related information.

(c) Update of the physics-related entity information.

Figure 3.6: Information flow in the various phases of the main simulation loop of ARGoS. The robot entities live in the global space. A controller and a set of sensors and actuators are associated to each robot. (a) In the initial phase, robot sensors collect information from the global space. Subsequently, robot controllers query the sensors and update the actuators with the chosen actions to perform. (b) The chosen actions stored in the actuators are executed, that is, the robot state is updated. At this point, positions and orientations have not been updated yet. (c) The physics engines calculate new positions and orientations for the mobile entities under their responsibility. Collisions are solved where necessary.

master thread sends the ‘start’ signal, the slave threads execute their work. Upon completion, the slave threads send the ‘finish’ signal to the master thread and switch back to idle state. When all the slave threads are done, the master thread sends the ‘start’ signal for the next phase.

The computation to be executed in each phase is decoupled in tasks assigned to the P slave threads. In the *sense+control* phase, a task is to update a robot’s sensors and then execute its controller. In the *act* phase, a task is to update a robot’s actuators. In the *physics* phase, a task is to update one physics engine. Two methods are available in ARGoS to assign tasks to threads. The first method is to pre-compute task assignment at the beginning of the experiment, distributing the tasks evenly among the threads. The assignment is kept constant unless robots are added or removed during the experiment. This method gives the best performance when the tasks performed by each thread have similar computational costs. This occurs, for instance, when all of the robots execute the same controller and the robots are evenly distributed across the physics engines. Otherwise, the duration of each phase of the loop corresponds to the time spent by the slowest thread to perform its part of the work. To solve this problem, a second method, called *h-dispatch* (Holmes et al., 2010), is available in ARGoS to assign tasks to threads. In this method, an additional thread called *dispatcher* is used. The dispatcher manages the list of tasks to perform at each phase. To perform a task, a thread receives it from the dispatcher. When the task is completed, the thread requests a new task until all the tasks have been performed. Then, the master thread sends a signal to the dispatcher to manage the tasks of the next phase, and the slave threads are awaken and resume fetching tasks. This method gives best performance when the tasks are very diverse. For example, if a slave thread happens to receive a long task from the dispatcher, the other slave threads can perform multiple shorter tasks in the meantime. However, if the tasks are very similar to each other, this method proves to be slower than pre-assignment, as the slave threads tend to finish at the same time and spend most of the time waiting for the dispatcher thread to assign new tasks to them. The choice of the thread assignment method is left to the user as a parameter in the experiment configuration file.

The scheduling strategy in ARGoS is implemented as a policy, analogously to spatial indexing. This design choice makes it possible to add new scheduling strategies to ARGoS. This choice is conceived to enable future research in optimal task scheduling, possibly including strategies capable of self-configuring depending on the type of experiment at hand.

Summary

In this chapter, I presented the ideas that underlie the design of ARGoS. From the discussion in Chapter 2, I derived a number of requirements that a successful design must respect: *(i)* accuracy, which is the degree of correspondence between results obtained in simulation and in real-world experiments; *(ii)* efficiency, in the form of static efficiency (i.e., the wise use of computational resources) and dynamic efficiency (i.e., the capability to maintain the simulation performance within acceptable bounds for the parameter range in which experimentation is performed); and *(iii)* flexibility, cast as the possibility to execute any kind of experiment, given the appropriate models. While flexibility is an intrinsic property of a simulator design, accuracy and efficiency are extrinsic, because they depend on the needs of the experimenter.

The first design choice to achieve the above requirements is modularity. Differently from the abstract, high-level nature of the simulator architectures discussed in Chapter 2, the architecture of ARGoS is decomposed into modules that map directly to the aspects a user might want to override, namely entities, sensors, actuators, physics engines, media, controllers, and even visualizations. For added flexibility, ARGoS offers a special kind of module, called 'loop functions,' that allows the user to interact with the simulation by injecting experiment-specific logic to derive statistics, modify the course of an experiment, or insert new functionality. Through the wise choice of modules, the user can tune the accuracy of the simulation, and at the same time enhance efficiency by assigning less computational resources to the aspects that do not influence its dynamics.

The second design choice of ARGoS stems from the fact that a large part of the computational resources is dedicated to managing spatial data. Many algorithms for spatial indexing exist, and their performance varies depending on the characteristics of the objects to handle. For instance, array-based methods offer best performance when a large number of similar objects is simulated; however, when the objects vary in size, tree-based methods tend to perform better. To ensure maximum performance, in ARGoS spatial indexes are cast as policies, and the user can select the most appropriate in a case-by-case fashion.

The third design choice is the possibility to partition the physical space into multiple regions, assigning each region to a dedicated physics engine. This feature of ARGoS distinguishes it among the existing simulators. By employing multiple physics engines, the simulation is executed faster, because each engine must handle a lower number of entities. Additionally, each engine can be specialized for a scenario, such as underwater experiments or navigation on a shore. In this way, complex experiments can be executed while, at the same time, maintaining the right level of accuracy for each aspect and ensuring efficiency.

The fourth design choice is multi-threading. In ARGoS, multi-threading is realized by considering the update of each part of a robot as a separate task. In this way, the granularity of the tasks is very thin, and parallelism is enhanced. The time taken by different tasks is typically very diverse, creating the problem of properly scheduling the execution of the tasks. ARGoS currently offers two scheduling algorithms: the first is designed to efficiently schedule tasks whose duration is similar; the second is designed to provide best results when the task duration is heterogeneous. Task scheduling is implemented as a policy, thus making room for future improvements.

Chapter 4

Achieving Flexibility

In Chapter 3, I presented an abstract view of the structure of ARGoS. I highlighted the fact that the ARGoS architecture is organized into a number of modules, each of which fulfills a specific purpose. From the architectural point of view, a user extension is essentially an instance of a specific module. The extremely flexible nature of ARGoS primarily derives from this design choice.

In this chapter, I present the approach adopted in ARGoS to realize the architectural view of Chapter 3. In Section 4.1, I illustrate requirements and desired features, arguing that realizing the ARGoS architecture is challenging. In Section 4.2, I focus on the basic aspects of the design of the ARGoS architecture. By "basic", I mean those aspects that do not constitute a challenge. In Section 4.3, I move my attention to the real design challenges, by *(i)* highlighting the shortcomings of a number of common approaches, and *(ii)* presenting the approach I followed in the design of the ARGoS architecture.

4.1 Requirements

Figure 4.1 reports a high-level representation of the required interactions among the ARGoS core, the developers of new modules, and the users that run simulations. As it can be inferred from the figure, realizing the architectural vision of Chapter 3 translates into designing mechanisms to achieve two purposes: *(i)* managing the available modules effectively, and *(ii)* separating the work of extension developers and end users. The design of these mechanisms are challenging because they must meet a number of diverse requirements that involve several aspects.

From the point of view of the design of the ARGoS core itself, two requirements are paramount: independence and efficiency. *Independence* means that the addition of new modules must avoid any modification of the ARGoS core. In other words, the ARGoS core must not depend on any specific module—although the implementation of a module depends

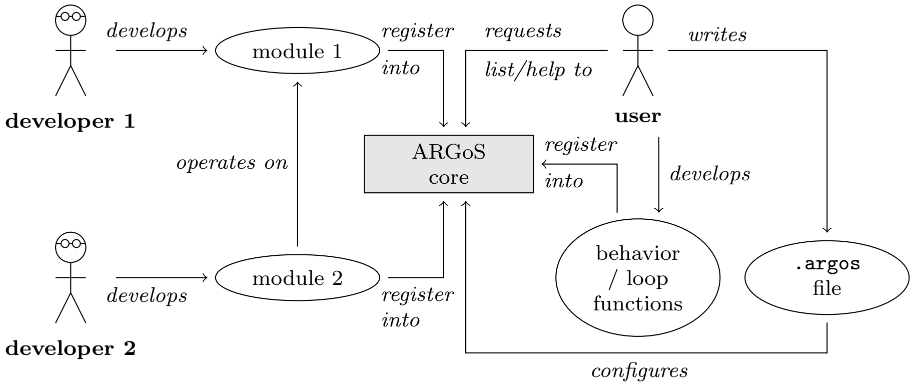


Figure 4.1: A high-level diagram representing the required interactions among developers, users, and the ARGoS core.

on the current version of the ARGoS core. *Efficiency* refers to the ability of the core to manage the life cycle of the different modules with minimum computational overhead.

From the point of view of the developer of extensions, flexibility and type safety are the main concerns. The different types of modules that compose the ARGoS architecture categorize the possible types of extensions ARGoS accepts. However, the actual logic contained in a module implementation is completely unforeseeable by ARGoS. To respect the requirements of efficiency and tunable accuracy, the ARGoS architecture must prove *flexible* enough to allow for highly optimized implementations of a certain module. In addition, often a certain module implementation involves *direct interactions with other modules*. It is important to notice that also the type of interaction between modules is unforeseeable by ARGoS, thus the core must provide a mechanism to enable arbitrary module-to-module interactions. For this kind of interactions to occur, besides flexibility ARGoS must ensure *type safety*, i.e., ARGoS must prevent incorrect interactions from happening. For instance, consider the interaction between entities and visualizations. Entities are added to the space, and the visualization has the task of representing the state of the space to the user, usually through graphics. To this aim, the visualization module must possess a specialized model for each type of entity, and display the right model accordingly to the state of each actual entity. To obtain its state, the visualization must interface directly with the entity—for independence, the ARGoS core cannot possess specific information on an entity. In this example, type safety is ensured if ARGoS prevents a visualization from rendering an entity with the wrong model. Also, in case a model for a certain entity type is missing, ARGoS must provide mechanisms to inform the user that the corresponding entities cannot be visualized. Without type safety, extension development would be cumbersome and error-prone, forcing the developer to include error-checking routines that are likely to lower the performance of an experiment run.

From the point of view of the user, *simplicity* is the key requirement. Operations such as choosing and loading a module should be performed with minimal effort. To ease the choice of the most suitable modules, the system must construct, manage, and present a list of available modules, along with a detailed usage guide. In addition, it is desirable that the mechanism to load experiment-specific logic (e.g., robot control code and loop functions) is analogous (or, preferably, identical) to the mechanism to load other modules, such as sensors and actuators.

Satisfying all of the above requirements is challenging. In the subsequent discussion, I divide the presentation of the final design in two parts. In Section 4.2, I illustrate the basic mechanisms to store modules and load them at the beginning of an experimental run. In Section 4.3, I criticize a number of classical techniques to enable arbitrary interactions among modules, highlighting their limitations with respect to the requirements above mentioned; subsequently, I explain the solution I employed to satisfy all the requirements.

4.2 Modules as Plug-ins

To satisfy the independence requirement, i.e., new extensions must not force a change into the ARGoS core, modules are implemented as *plug-ins*. Plug-ins are the classical approach to enable third-party additions to a software without modifying its architecture nor requiring recompilation (independence). Plug-ins are software libraries that can be linked (and unlinked) at any moment during the life-time of the target software. In modern computing, plug-ins constitute the backbone of many programs of widespread use. For instance, modern web browsers use plug-ins to visualize special content such as Flash videos; media players encapsulate audio/video codecs into plug-ins; software development environments support additional programming languages through extensions implemented as plug-ins.

In this section, I will sketch the design of the plug-in manager of ARGoS. In Section 4.2.1, I explain how the various plug-in types are organized. In Section 4.2.2, I describe the actual plug-in manager.

4.2.1 The Plug-in Hierarchy

Regardless of its intended task, the life-time of a plug-in is usually composed of at least three phases: initialization, execution, and destruction. Initialization consists in performing the necessary duties for the execution to occur. Typically, configuration and memory allocation are

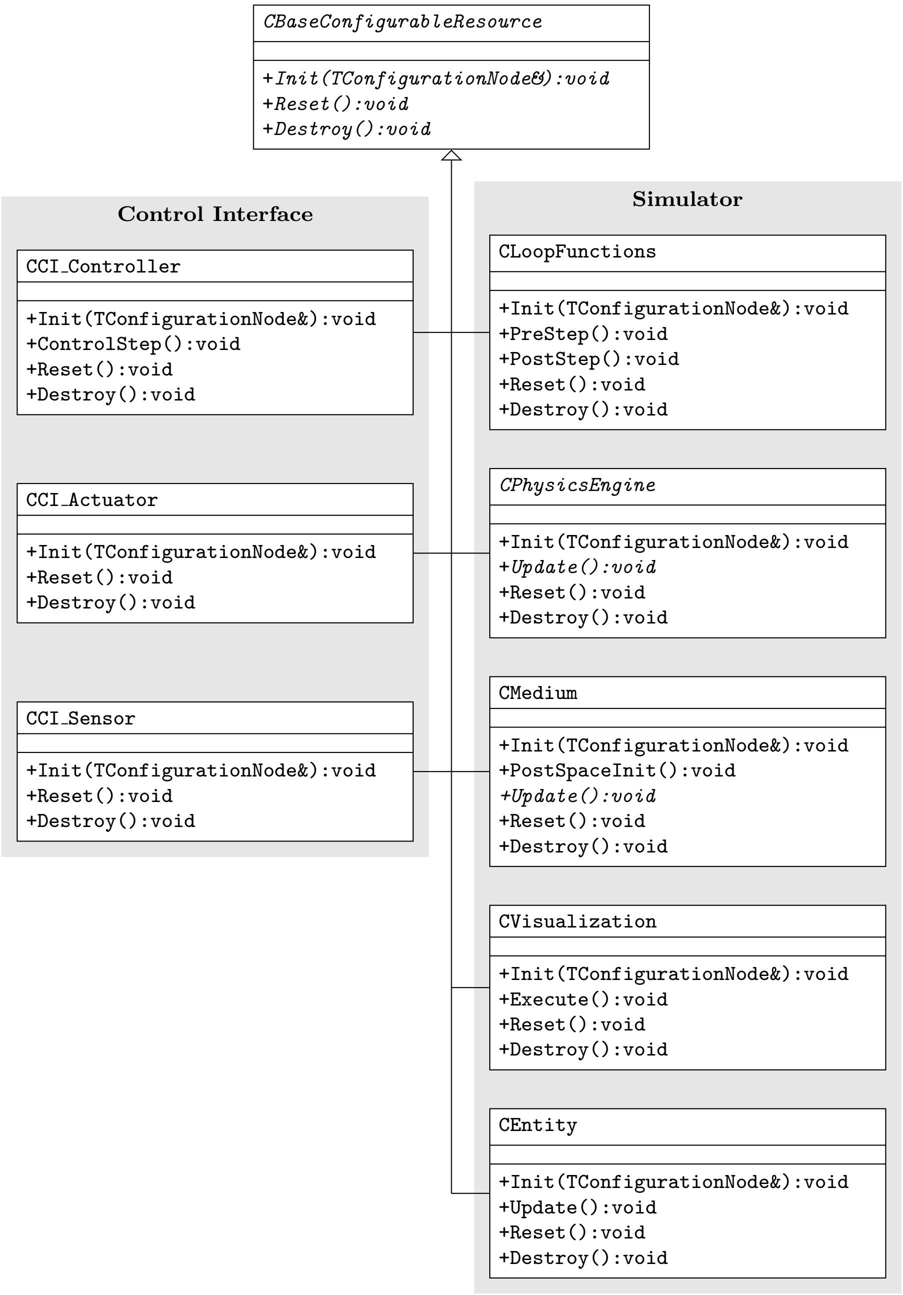


Figure 4.2: A UML class diagram of the basic plug-in hierarchy in ARGoS.

performed during this phase. The execution phase contains the actual logic of the plug-in. Destruction reverts the actions performed during initialization, such as disposing of memory and closing open files. In ARGoS, beyond these phases, a fourth one is present—reset. The reset phase has the task of reverting the state of the simulation to what it was right after initialization.

Figure 4.2 depicts a class diagram of the plug-in hierarchy of the ARGoS core. The basic plug-in type is `CBaseConfigurableResource`, which contains three methods—`Init()`, `Destroy()`, and `Reset()`. These methods execute the corresponding three phases in the plug-in life cycle. The fourth phase, execution, is not present in the base class. I made this choice for two reasons.

First, because the execution logic of a plug-in is not always contained in one method. For instance, the execution logic of the `CLoopFunctions` class is split in two methods, `PreStep()` and `PostStep()`, executed right before and right after a complete simulation update step has occurred. Also the `CCI_Sensor` and `CCI_Actuator` classes require special treatment. As depicted in Figure 3.1, the purpose of the control interface is to abstract away the fact that the underlying robot is simulated or real. The control interface, in fact, is composed of a set of interfaces that inherit from the `CCI_Sensor` and `CCI_Actuator` classes. The "real" work is performed by suitable implementations of these classes. On the real robot, a sensor's job is to *collect* data from the robot; in a simulation, a sensor's job is to *calculate* the same data. Similarly, on the real robot, an actuator must use the *platform primitives* to execute the actions dictated by the control logic; in a simulation, an actuator must *modify the robot state*. On the real robot, the implementation of these classes depends on the internals of the robotic platform. Often, data between the control interface and the robot subsystems is exchanged *asynchronously*, either through interrupts (e.g. the e-puck) or message passing (e.g., the foot-bot). In this case, a single update method is useless. In the simulator, on the other hand, data is managed *synchronously*, which calls for an explicit update method.

The second reason for not having an explicit execution method in the base plug-in class is that it is desirable, for readability purposes, that the name corresponding to the execution method is somehow related to the kind of logic it is supposed to contain. For instance, `ControlStep()` is a good choice for the `CCI_Controller` class, and `Update()` fits well the purposes of the `CPhysicsEngine` and `CEntity` classes.

4.2.2 Plug-in Management

In general, the plug-in manager is a software component that performs two basic tasks: *(i)* allowing the system to create new instances of a given plug-in when necessary; and *(ii)* registering new plug-in implementations.

Regarding the creation of new plug-in instances, the classical approach is to employ the *Factory Method* pattern (Gamma et al., 1994). This design pattern tackles the problem of creating an instance of a class without knowing its interface. The main idea is to provide an interface to create an object, and defer the actual operations to perform this task to a type-specific implementation. Since in ARGoS many plug-in types exist, I slightly generalized this pattern (see Section 4.2.3). As an added benefit in terms of independence and flexibility, the resulting system can be exploited by an extension developer or by the end user to create further plug-in types with only a couple of lines of code.

In order to register a new plug-in, the plug-in manager requires not only its implementation, but also a certain quantity of metadata. In the Eclipse IDE,[1] for example, each plug-in is bundled with an XML file called *manifest* that contains the human-readable name of the plug-in, its version, details about the functionality it implements, as well as a link to the actual plug-in binary. Upon initialization, the Eclipse core spans the plug-in directories, opens each bundle, and retrieves the information on every available plug-in. This system is effective and widespread in plug-in-based software. However, its main drawback is forcing the core to open and parse a large number of text files upon initialization, which results in long start-up times when many plug-ins must be scanned. To avoid this issue, for ARGoS I chose to include the metadata directly into the plug-in binary. This approach, explained in Section 4.2.3, has the benefit that the metadata is pre-parsed and automatically loaded into the ARGoS core, making initialization and help queries extremely fast.

4.2.3 The Plug-in Factory

The class diagram of the plug-in manager is reported in Figure 4.3.

The `CFactory` template is the core of the plug-in manager. Its interface offers three methods—`GetTypeMap()`, `Register()`, and `New()`, to query the available plug-ins, register a new plug-in, and create an instance of a specific plug-in, respectively. Internally, `CFactory` uses a hash map of `STypeInfo` structs to store the plug-in metadata and a function pointer to the plug-in creator. In the hash map, each struct is associated to a string label that identifies a specific plug-in implementation. `CFactory` is a static class because, besides the hash map, it does not need to maintain any state information.

`CFactory` is a template whose parameter `T` corresponds to the base of the plug-in hierarchy (e.g., `CCI_Controller`, `CPhysicsEngine`, etc.),

[1] http://www.eclipse.org/

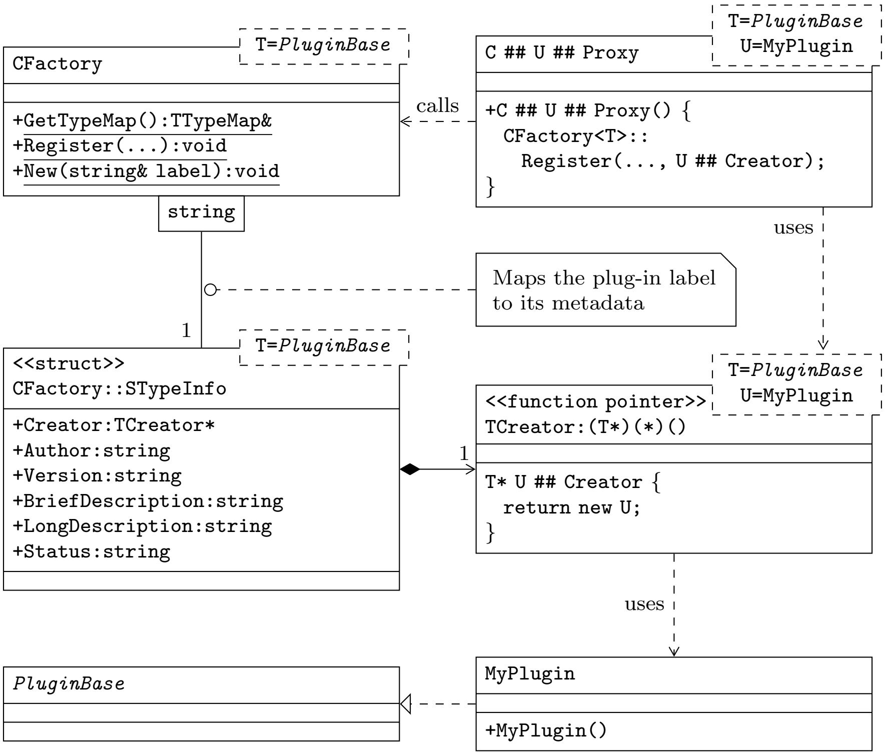


Figure 4.3: A UML class diagram of the ARGoS plug-in manager. In this diagram, I use the notation `A ## B` to express the string concatenation operator of the C++ pre-processor. In the C++ pre-processor, the expansion of a symbol `X` corresponds to its definition, if the symbol was previously defined; otherwise its expansion corresponds to the symbol itself. Thus, `A ## B` is the string resulting from the concatenation of the expansions of `A` and `B`.

indicated in the diagram as ***PluginBase***. The choice of using a template class is a generalization of the classical Factory Method pattern. In the latter, `CFactory` is a class rather than a template, because the Factory Method pattern is designed to create plug-ins from a *single hierarchy* whose base is known *in advance*. However, in ARGoS, several plug-in types exist (see also Figure 4.2). If one were to follow the classical pattern strictly, the interface of the `CFactory` class would have to include dedicated methods `GetTypeMap()`, `Register()`, and `New()` for each plug-in type. Thus, whenever a new plug-in type is added to ARGoS, the interface of `CFactory` would require modifications. By making `CFactory` a template whose parameter is the base of the plug-in hierarchy, these modifications are not necessary, and `CFactory` can be used to create plug-ins from *multiple hierarchies* whose bases are *not known in advance*. The benefits in terms of independence and flexibility are significant—the plug-in manager can even be used by extension developers and end-users to add custom plug-in types independently from the ARGoS core.

Plug-in Creation

The creation of a plug-in instance occurs through the `CFactory<T>::New()` method.

This method accepts as a parameter the identifying label of the wanted plug-in implementation. Using this label, the method searches in the hash map for the corresponding `CFactory::STypeInfo` struct.

The latter contains a function pointer (`TCreator*`) to the plug-in *creator function*. As its name suggests, the role of the creator function is to return a new instance of the wanted plug-in. For each type of plug-in, there exists a dedicated creator function, registered in the system as explained in Section 4.2.3.

Plug-in Registration

The registration of a plug-in in ARGoS occurs through a mechanism called *auto-registration*. This mechanism exploits the way Unix-compatible systems link the global symbols of a library into an executable to piggy-back a call to `CFactory<T>::Register()`.

The operation of linking a library into an executable is composed of a few phases. One of the earliest phases is the so-called *static initialization*. During this phase, the linker collects information about the global symbols and, in particular, creates the global variables present in the library. Normally, the creation of a global variable only implies allocating memory. However, if the global variable is a class, it is also possible to include arbitrary code in its constructor.

The auto-registration mechanism consists in including a call to `CFactory::Register()` within the constructor of a suitably defined class declared as global variable. The class is defined specifically for the plug-in implementation to register. In the class diagram of Figure 4.3, the plug-in implementation class is `MyPlugin`. Thus, the registration class is named `CMyPluginProxy`, which is the result of applying the `##` operator to `C ## U ## Proxy` when `U` is set to `MyPlugin` (see the figure caption for an explanation on the `##` operator).

In addition to the registration class, one must also define a plug-in-specific creator function. In Figure 4.3, the function is called `MyPluginCreator`, result of the operation `U ## Creator`.

The above actions are encapsulated into a macro called `REGISTER_SYMBOL`. For the extension developer and the end user, registering a new plug-in implementation is merely a matter of calling this macro.

Discussion

The registration of plug-in implementations and the creation of new instances is a well-understood problem.

A slight generalization of the classical Factory Method pattern solves the problem of creating new plug-in instances, when multiple plug-in hierarchies exist whose base classes are not known in advance. From the point of view of an extension developer, this solution guarantees extreme flexibility; from the point of view of the ARGoS core design, the independence requirement is fully respected.

Registering new plug-in implementations is an equally simple problem to solve, provided some knowledge of the internals of the Unix operating system. The final result is a system that can register any plug-in using the same underlying machinery. In addition, from the point of view of the end user, the simplicity requirement is respected, because the internals of the registration logic are completely encapsulated in a single macro definition.

4.3 Arbitrary Interactions among Modules

The requirement of allowing for arbitrary interactions among modules mainly stems from the particular role that entities play with respect to the other modules. In fact, entities store part of the state of a simulated object. Other modules, such as physics engines and visualizations, access the entities to perform their work—modifying the entity state or reading it. In other words, entities act as a *mediator* between other module types. As discussed in Chapter 3, this design choice results in a clear factorization of responsibilities among different modules, which, in turn, makes it possible to achieve multi-threading without introducing race conditions.

4.3.1 Issues

A General View

The main issue in the realization of this design choice is that entities are modules themselves, because developers must be given the possibility to add new ones (e.g., robots or other objects) at will.

To better understand this issue, refer to Figure 4.1. In the figure, I schematize the case in which two developers work separately. Developer 1 implements module 1, and developer 2 implements a different module, module 2, which interacts with module 1. A third independent party, the user, downloads both modules and employs them in an experiment. For the sake of explanation, module 1 corresponds to a new robot, and module 2 corresponds to a new physics engine. Since I assume that the two developers work separately, developer 2 does not know in advance

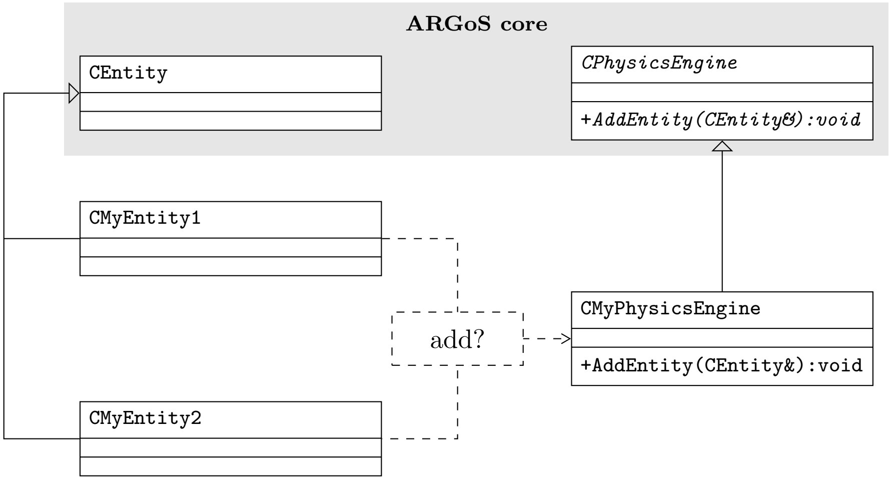


Figure 4.4: A UML class diagram for an example problem of inter-module communication. In this diagram, two entities must be added to a physics engine. The entities and the physics engine are implemented by three different developers, out of the ARGoS core.

the features of the robot implemented by developer 1. Thus, the easiest solution for developer 2 is to provide an API to create physics models for robots, and leave the task of providing the models to other people. Conversely, since an entity is supposed to be a mediator between modules coded by several people, developer 1 must be able to define the features of the robot entity without having a specific module in mind. In other words, in this example, the problem is how to associate a physics model for a specific physics engine to a generic robot entity, when the two parties have been developed by different people. To make the picture ever more ambitious, the simplicity requirement dictates that these implementation details are hidden from the user, who just wants to use the modules without knowledge of their internals; and the efficiency requirement pushes towards a solution with minimal computational overhead.

It is easy to imagine that the machinery to realize arbitrary interactions between entities and other modules could be useful to enable other inter-module interactions, too. A visualization, for instance, could interface directly with a specific sensor implementation to visualize extra data for debugging purposes.

Common Implementation Approaches

For the sake of explanation, in this section I will consider the interactions between entities and physics engines, but the discussion is applicable to any kind of module. Figure 4.4 reports a UML diagram of the example.

In terms of classes, the problem of inter-module communication can be expressed as follows. The ARGoS core maintains a number of data structures, each dedicated to a specific module type. In particular, ARGoS maintains a list of pointers to `CEntity` objects and a list of pointers to `CPhysicsEngine` objects. Among the many interactions between these two kinds of modules, consider the addition of an entity to a physics engine. To achieve this, the physics engine must create an internal representation of the entity body. Thus, the physics engine must know the exact type of the entity being added. However, the types of both the physics engine and the entity are defined outside the ARGoS core and known only at run-time. Consequently, the logic that adds a specific entity to a specific physics engine must be selected at run-time. In computer science, this problem is often referred to as *dynamic dispatch* (Lippman, 1996).

ARGoS is written in C++, because this language offers a good balance between abstraction and performance. However, C++ has only partial support for dynamic dispatching—*single dispatching*.[2] Single dispatching refers to the capability of the language to select the right implementation of a method based on the type of the class to which the method belongs. This mechanism is one of the cornerstones of polymorphism in C++. For the problem at hand, polymorphism (and, thus, single dispatching) is not sufficient. In essence, polymorphism refers to the possibility to use a single interface for different objects, and to call the right method implementations depending on the run-time type of the objects. In the example of Figure 4.4, this means that if the interface `CPhysicsEngine` exposes a public method `AddEntity(CEntity&)`, and `CMyPhysicsEngine` implements `AddEntity(CEntity&)`, then C++ can select `CMyPhysicsEngine::AddEntity(CEntity&)` from a pointer to `CPhysicsEngine`. This mechanism does not achieve the complete result because it is necessary to also dispatch over the argument of the `CMyPhysicsEngine::AddEntity()` method to select the right entity type. To work around this limitation of C++, several techniques exist.

The Observer Pattern. In the Observer pattern (Gamma et al., 1994), an object, called the *subject*, maintains a list of other objects, called the *observers*. Whenever its state changes, the subject notifies the observers of the fact. This pattern is often employed to implement event handling mechanism, such as signal/slot. For the problem at hand, this pattern can be applied in two alternative ways. First, by equipping the `CEntity` class with a method for each possible operation a module can perform on it. This solution is not acceptable, because this would cre-

[2] Although an extension to support multiple dispatching has been proposed by Pirkelbauer et al. (2007)

ate a dependency between the ARGoS core (the `CEntity` class) and the user-defined modules. The second way would consist in equipping the `CPhysicsEngine` class with a method for each operation it can perform on entities. The problem with this solution is that, in the Observer pattern, the subject notifies *all* of its observers. Thus, a further mechanism should be added to identify which entities must be set as observers—a mechanism that would be dependent on the entity type, once more breaking independence.

The Mediator Pattern. The Mediator Pattern aims to simplify the management of the interactions among multiple classes (Gamma et al., 1994). In this pattern, the designer defines a dedicated class type that encapsulates the necessary interactions. The Mediator pattern, rather than tackling a dispatching problem, captures the situation in which a relatively large number of classes interact in complex ways. In other words, the mediator decouples the design of the interacting classes. To be able to perform its work, however, the mediator must know the exact interface of the interacting classes at compile time—which is precisely the missing information for the problem at hand.

The Visitor Pattern. The aim of the visitor pattern is to separate the definition of the operations from the object structure on which they are applied (Gamma et al., 1994). In particular, the visitor pattern allows developers to add new operations to an existing object structure, without modifying it. In the example at hand, the visitor pattern is realized adding a function `Accept(CAbstractOperation& o)` to the `CEntity` class. `CAbstractOperation` is the base class of an hierarchy of user-defined operations, and it consists of a list of methods `Visit(CMyEntity1&)` ...`Visit(CMyEntityN&)`. Each entity implements `Accept(CAbstractOperation& o)` with a single line of code: `o.Visit(*this)`. In essence, this technique is a way to achieve double dispatching: the first dynamic dispatch occurs when `CEntity::Accept()` is called, and the second occurs when `o.Visit(*this)` is executed. This method works best when the object structure on which the operations are applied is *(i)* known a priori, because the object structure is necessary to define the class `CAbstractOperation`, and *(ii)* stable, because any modification to the object structure entails a modification of `CAbstractOperation` and each of its descendants. For the problem at hand, none of these conditions is satisfied.

4.3.2 Solution: The Cooperative Visitor

The core of the problem of generic inter-module communication is that the selection of a specific operation depends on the run-time properties of both modules involved. As mentioned, this problem can be solved through multiple dispatching. In particular, since dispatching occurs over two module types, what is necessary is a technique that achieves double dispatching.

The Visitor pattern, as discussed, offers a way to achieve double dispatching, but it creates a dependency between the ARGoS core and the modules. For the design of ARGoS, I chose to pursue an approach that generalizes the Visitor pattern, respects the independence requirement, and even allows for further extensions. This approach is called *Cooperative Visitor* (Krishnamoorthi, 2007).

The combination of the Cooperative Visitor with the plug-in management mechanisms explained in Section 4.2 is rather complex. In this section, I will focus only on the most important aspects of the design. I encourage the interested reader to refer to (Krishnamoorthi, 2007) for an introduction to the Cooperative Visitor, and to inspect the commented code in the following files:

- `argos3/src/core/simulator/entity.h`, which contains the definition of the basic entity operation, its management, and its registration logic;
- `argos3/src/core/utility/vtable.h`, which contains the definition of the tagging engine and the virtual table (see below);
- `argos3/src/plugins/simulator/physics_engines/dynamics_2d/dynamics_2d.h`, which contains an example of use of the Cooperative visitor to manage the addition of entities into ARGoS' 2D dynamics engine.

4.3.3 Operations

An operation is represented as a class that contains a single method, `ApplyTo()`, with two arguments: a pointer to the *context* and a pointer to the *operand*. The context, in our example, is `CMyPhysicsEngine`, and the operand is either `CMyEntity1` or `CMyEntity2`.

Analogously to plug-ins, operations are registered into ARGoS through a dedicated macro. Upon registration, an operation is stored in a data structure. This data structure enables the ARGoS core to call the operation without knowing the run-time properties of the modules involved.

To hide the types of the context and of the operand, an operation is wrapped into an *adapter* (Gamma et al., 1994). The adapter is a class that exposes an interface based on information known to ARGoS (e.g., `CEntity` and `CPhysicsEngine`) and that internally performs the necessary transformations to execute the wanted operation. In other words, the adapter internally performs the double dispatch.[3] In the example,

[3] In the jargon of compiler design, an adapter with this behavior is referred to as *thunk*.

the first dispatch is the transformation of the pointer to `CPhysicsEngine` into `CMyPhysicsEngine`; the second dispatch is the transformation of the pointer to `CEntity` into `CEntity1` or `CEntity2` (depending on which is the operand). Typically, these transformations must be performed through the type-safe (but slow) `dynamic_cast` C++ command. However, due to the guarantees offered by the data structure that manages the operations (described below), it is possible to spare the cost of `dynamic_cast`, and use the faster `static_cast` command instead, thus enhancing efficiency sensibly.

The adapter is implemented as a class template that takes as parameters the context, the operand, and the return type of the `ApplyTo()` method. This ensures flexibility, because not only the adapter can be used with any kind of module, but it also allows the developer to define operations that return arbitrary types.

Storing and Retrieving an Operation

The Cooperative Visitor is based on the idea that operations are stored into a structure that associates each operation to a unique tag. Krishnamoorthi (2007), following the jargon of compiler design, refers to this structure as *virtual table*. Since ARGoS cannot know the run-time type of the operations, the virtual table actually associates a tag to the adapter of an operation.

At any moment during the execution of ARGoS, multiple virtual tables exist. Each virtual table refers to a particular type of operation. For instance, a virtual table exists to store all the operations that add an entity to `CMyPhysicsEngine`. The creation of a virtual table occurs at run-time as soon as ARGoS registers a new operation for a module. If the table does not already exist, it is created on-the-fly and the operation is added. Subsequent operations of the same type are stored in this virtual table. To identify a certain type of operation (not an actual operation), the developer is required to provide any valid C++ symbol upon registering an operation. The symbol is internally used as a label for the table. In practice, since operations in ARGoS are typically structured into a hierarchy, the base class of such hierarchy is employed to label a virtual table.

As explained, each virtual table associates a tag to an operation. Since a virtual table collects all the operations of the same type (e.g., adding `CMyEntity1` or `CMyEntity2` to `CMyPhysicsEngine`), a tag must identify the operand (e.g., `CMyEntity1` and `CMyEntity2`). To this aim, the Cooperative Visitor employs a template-based tag engine that assigns a unique integer to all classes in a hierarchy. The base class (e.g., `CEntity`) is assigned the value 0, and subsequent entities, upon registration, are automatically assigned increasing (an unique) values. Using integers as tags is a convenient design choice because the virtual table can be implemented as an array of adapters. Besides its simplicity, this approach is also efficient both in terms of time complexity (O(1)) and space complexity (O(N), where N is the number of registered entity classes).

Executing an Operation

The selection and execution of an operation follows a number of phases.

For instance, when a robot is added into the ARGoS space, its embodied entity component is also added. Upon adding the embodied entity, ARGoS identifies which physics engine is in charge of managing it. Subsequently, ARGoS calls the `AddEntity(CEntity&)` method of the physics engine. At this stage, ARGoS has used a pointer to `CPhysicsEngine` to call `AddEntity()`, and the argument passed to this method is a reference to a `CEntity`.

Through polymorphism, C++ can dispatch the call to `AddEntity()` to `CMyPhysicsEngine::AddEntity()`. The implementation of this function, upon receiving as argument a reference to `CEntity`, cannot rely on any information on the specific type of entity received. Thus, it must employ the cooperative visitor.

To this aim, the physics engine calls a convenience macro offered by ARGoS. This macro retrieves the virtual table corresponding to the type of operation to execute, e.g., adding an entity to `CMyPhysicsEngine`. Next, it retrieves the tag of the entity, and uses it to obtain a pointer to the adapter of the wanted operation. With the adapter, the macro can execute the operation passing as arguments a pointer to itself (context) and a pointer to the entity (operand). The adapter performs the double dispatch and calls the operation. Finally, the operation performs the necessary activities to insert the right model into the wanted physics engine.

Discussion

The Cooperative Visitor satisfies all of the requirements. It ensures

- independence, through the adapter;
- efficiency, through the use of `static_cast` for type conversion and integer tags to identify the entities;
- type safety, through the use of unique identifiers for entities and labels to categorize operations into virtual tables; and
- simplicity, because this mechanism is completely hidden to the user, and exported to the developer as a set of easy-to-use macros.

In addition, with the Cooperative Visitor it is simple to handle cases such as missing operations for a certain entity type. It is enough to associate a default operation for the `CEntity` class that provides the wanted logic.

Finally, the implementation of the virtual table and of the tagging engine is completely generic. It is not designed to work specifically with entities. Consequently, it possible to reuse the same mechanisms to enable arbitrary interactions among modules. In this way, for instance, a visualization can display debugging information on the data of a sensor.

Summary

This chapter was devoted to the low-level implementation aspects of the ARGoS architecture. In order to realize the high-level description in Chapter 3, a number of requirements must be satisfied. From the point of view of the ARGoS core, the mechanisms to handle the modules must ensure independence (i.e., the core does not depend on any specific module implementation) and efficiency (i.e., the computational cost of handling modules must be as low as possible). From the point of view of a third-party developer extending ARGoS, the architecture must prove flexible: It must allow for any kind of extension, and it must enable generic and type-safe inter-module communication. This is particularly important because entities, in ARGoS, act as mediators among the other modules. Finally, from the point of view of the user, configuring an experiment, selecting modules, and executing the code must prove simple. To meet these requirements, I made two design choices.

First, modules are implemented as plug-ins, i.e., software modules loaded at run-time. Plug-ins are organized into a simple hierarchy structured to offer both abstraction and flexibility. Abstraction is achieved by fixing a shared life-cycle for any module type (i.e., init/update/reset/destroy); flexibility is ensured by leaving the specification of the update phase of the life-cycle to the specific module type. The creation and destruction of plug-ins is performed by ARGoS through a general-purpose factory, available to the user for further extensions.

Second, to enable arbitrary interactions among modules in a type-safe manner, I employed the Cooperative Visitor. This design pattern is a generalization of the well-known Visitor Pattern that offers the same guarantees of the latter in terms of type-safety, efficiency, and generality. In addition, the Cooperative Visitor removes the primary defect of the Visitor Pattern—the dependency between the module implementation and the ARGoS core. The resulting system is flexible, efficient, and easy to use.

Chapter 5

Efficiency Assessment

In this chapter, I discuss the experimental evaluation I conducted to assess the efficiency of ARGoS. To ease the analysis of the results, the experiments do not involve multiple types of robots, but rather employ a single type. I point the reader interested in experiments with ARGoS that employ different types of robots to Chapter 7 and to the following papers: (Ducatelle et al., 2010, 2011; Mathews et al., 2010; Montes de Oca et al., 2010; Pinciroli et al., 2009).

The chapter is organized as follows. In Section 5.1, I present the experimental setup. In Section 5.2, I discuss the results obtained with the 2D-dynamics physics engine. In Section 5.3, I analyze the results obtained with the other two physics engines available in ARGoS, the 2D-kinematic and 3D-dynamics engines.

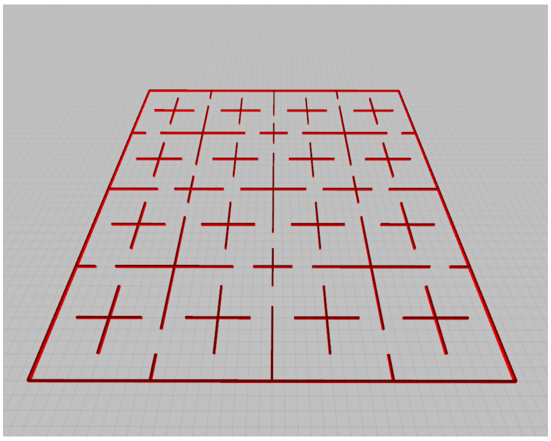


Figure 5.1: A screen-shot from ARGoS showing the simulated arena created for experimental evaluation.

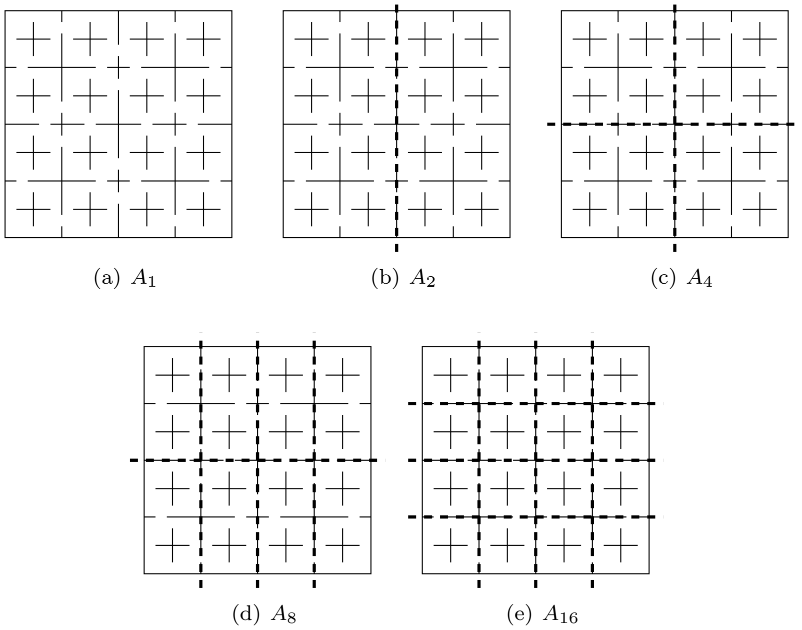
5.1 Experimental Setup

So far, little work has been devoted to characterize the performance of simulators for more than a few dozens of robots. The only remarkable exception is offered by Stage's performance evaluation by Vaughan (2008). Vaughan proposes a benchmark experiment in which thousands of wheeled robots diffuse in a large environment while avoiding obstacles. Despite its simplicity, this benchmark is appropriate because it tests the core functionality of a simulator (mostly ray casting and collision-related calculations). In addition, the robots perform a meaningful task for the swarm robotics community.

To test ARGoS against this benchmark, I setup an arena that mimics an indoor environment (see Figure 5.1). The arena is a square whose sides are 40 m long. Similarly to Stage's benchmark, I employed the simplest wheeled robot available in ARGoS, the e-puck (Mondada et al., 2006). Each robot executes a simplified version of the diffusion algorithm proposed by Howard et al. (2002).

I assess performance with two standard measures: wall clock time and speedup. *Wall clock time*, hereinafter denoted by w, is a measure of the time elapsed between an experiment's start and its completion. Wall clock time is typically affected by the quantity and the type of other applications running simultaneously on the same machine. For the purposes of this analysis, I conducted the experiments on dedicated machines, in which I limited the running processes to those necessary for the normal execution of the operating system and ARGoS. The second performance measure I employ is the *speedup*, denoted by u. To obtain this measure, I first consider the total CPU time (denoted by c) of the process running ARGoS. Such time differs from the wall clock time in that the CPU time increases only when the process is actively using the CPU. On multi-core CPUs, I obtain a measure c_i for each core. Thus, the total CPU time c is given by the sum of the c_i: $c = \sum_i c_i$. The speedup is then calculated as the ratio between the total CPU time and the wall clock time: $u = c/w$. Intuitively, the speedup measures the extent to which parallelism was exploited by the process during its execution. Therefore, in single-core CPUs or in single-threaded applications, $u \leq 1$. In contrast, in multi-threaded applications running on multi-core CPUs, the objective is to obtain speedup measures significantly greater than 1.

In the analysis, I focus on three factors that strongly influence performance: *(i)* the number of robots N, *(ii)* the number of parallel slave threads P, and *(iii)* the way the environment is partitioned into regions, each assigned to a different physics engine. Concerning the number of robots, I conducted experiments with $N = 10^i$, where $i \in \{0, 1, 2, 3, 4, 5\}$. To test the effect of the number of threads P, I run the experiments on



(a) A_1

(b) A_2

(c) A_4

(d) A_8

(e) A_{16}

Figure 5.2: The different space partitionings (A_1 to A_{16}) of the environment used to evaluate ARGoS' performance (a screen-shot is reported in Figure 5.1). The thin lines denote the walls. The bold dashed lines indicate the borders of each region. Each region is updated by a dedicated instance of a physics engine.

four machines with 16 cores each,[1] and let $P \in \{0, 2, 4, 8, 16\}$. When $P = 0$, the master thread executes all tasks without spawning the slave threads. Finally, I define five ways to partition the environment among multiple physics engines, differing from each other in how many engines are used and how they are distributed. I refer to a partitioning with the symbol A_E, where $E \in \{1, 2, 4, 8, 16\}$ is the number of physics engines employed. E also corresponds to the number of regions in which the space is partitioned. The partitionings are depicted in Figure 5.2. For each experimental setting $\langle N, P, A_E \rangle$, I ran 40 experiments. The simulation time step is 100 ms long. Each experiment simulates $T = 60\,\mathrm{s}$ of virtual time, for a total of 600 time steps. In order to avoid artifacts in the measures of w and u due to initialization and cleanup of the experiments, the measures of wall clock time and speedup include only the main simulation loop. For the same reason, I conducted the experiments without graphical visualizations.

In the rest of this chapter, I discuss the results I obtained using different types of physics engines. In Section 5.2, I focus on the re-

[1] Each machine has two AMD Opteron Magny-Cours processors type 6128, each processor with 8 cores. The total size of the RAM is 16 GB.

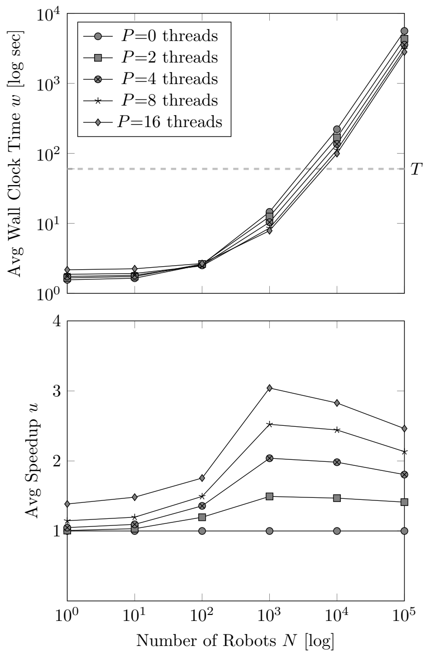


Figure 5.3: Average wall clock time and speedup for a single physics engine (A_1). Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, P, A_1 \rangle$. Each experiment simulates $T = 60\,\text{s}$. In the upper plot, points under the dashed line signify that the simulations were faster than the corresponding real-world experiment time; above it, they were slower. Standard deviation is omitted because its value is so small that it would not be visible on the graph.

sults obtained with ARGoS' 2D-dynamics engine. In Section 5.3, I discuss the result obtained with other two engines: 2D-kinematics and 3D-dynamics.

5.2 2D-Dynamics Physics Engine

In the first set of experiments, I employ ARGoS' 2D-dynamics engine. This engine is based on the scalable, state-of-the-art library Chipmunk Physics, which is widely used in both scientific applications and games. For more information on Chipmunk Physics, refer to Section 2.2.2.

5.2.1 Single Engine

In this section, I discuss the results of the experiments in which one physics engine updates all *embodied entities* in the arena (partitioning A_1).

The results are reported in Figure 5.3. I plot the average over 40 experiments of the wall clock time and the speedup for different values of N and P. The graphs show that multi-threading has beneficial effects on performance when the number of robots is greater than 10^2. Performance improves as the number of threads is increased, and the lowest wall clock time is measured when $P = 16$. In particular, let us focus on $N = 10^5$ and take as baseline the wall clock time measured when no threads are employed. Comparatively, with 16 threads ARGoS is twice as fast.

Moreover, when threads are used, speedup is always greater than 1. Its maximum, whose value is approximately 3.04, corresponds to $P = 16$ and $N = 10^3$. The observation that speedup decreases after hitting the maximum can be explained by the fact that only one physics engine is responsible for all *embodied entities*. Thus, in the *physics* phase, only one thread runs the physics engine, while the others are idle, not contributing to the measure of the CPU time c.[2] The more robots are employed in the simulation, the longer the thread updating the physics engine must work, while the others remain idle. Thus, the one thread in charge of physics increasingly dominates the measure of the speedup, worsening it as the number of robots grows.

As the plots illustrate, the threads negatively impact performance when $N < 10^2$. Profiling data revealed that, with few robots, the time spent on updating the robots and physics engine is comparable to the time taken by the master threads to manage the slave threads. Therefore, with few robots, it is better to avoid such overhead and let the master thread perform all the work. Supporting evidence for this explanation is offered by the very low values measured for speedup. With $N = 1$ and $P = 2$, the speedup is approximately 1 and with $P = 16$ it is 1.39.

5.2.2 Multiple Engines

In this section, I discuss the results of experiments in which the arena is partitioned into multiple regions managed by different physics engines. The results are showed in Figure 5.4.

A first important result is that the use of two physics engines, corresponding to space partitioning A_2, is already sufficient to perform a simulation of 10^4 robots that runs at the same rate as the corresponding real-world swarm. This result is reached when the maximum number of threads is utilized, $P = 16$. The trends of w and u with respect to the number of threads are qualitatively identical to those of partitioning A_1. Employing more threads results in increasingly better performance when $N > 10^2$ and the speedup is best for $P = 16$. Comparing wall clock times with $N = 10^4$ and $P = 16$ for partitioning A_2 and A_1, I obtain that $w(A_2)/w(A_1)$ is about 0.6.

[2] However, the *sense+control* and *act* phases are still executed in parallel.

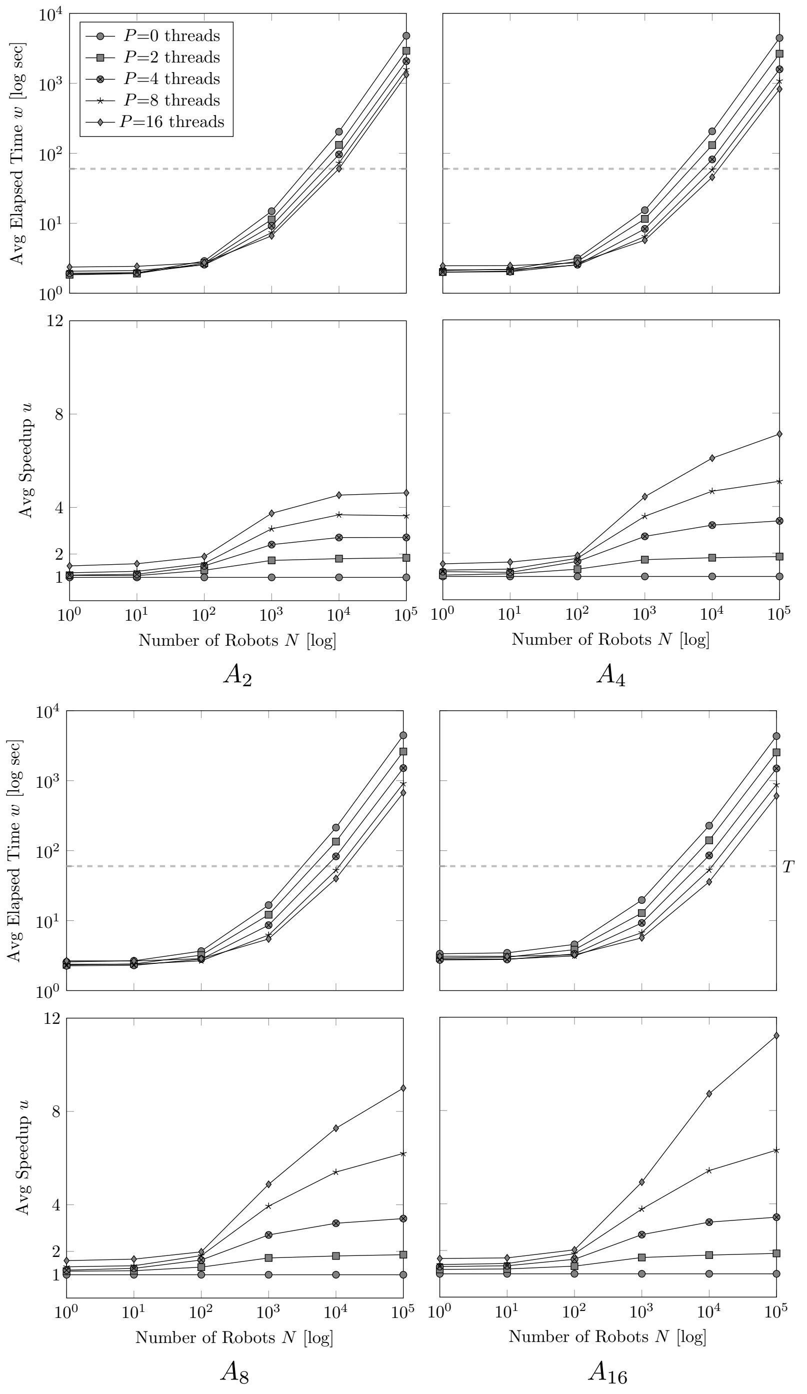


Figure 5.4: Average wall clock time and speedup for partitionings A_2 to A_{16}. Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, P, A_E \rangle$. Each experiment simulates $T = 60\,\text{s}$. In the upper plots, points under the dashed line signify that the simulations were faster than the corresponding real-world experiment time; above it, they were slower. Standard deviation is omitted because its value is so small that it would not be visible on the graph.

Performance constantly improves with the number of engines and of threads. The most remarkable result in Figure 5.4 is obtained for A_{16}, $N = 10^4$ and $P = 16$. In this configuration, the measured wall clock time is about $0.6T$, which means that ARGoS can simulate 10^4 robots in 60% of the corresponding real-world experiment time. For numbers of robots from $N = 10^3$ to $N = 10^5$ wall clock time grows roughly linearly. Moreover, for $N = 10^5$, wall clock time is about $10T$. Regarding speedup, the more robots are employed, the more parallelism is efficient. In the experiments, the largest speedup was observed with $P = 16$, A_{16} and $N = 10^5$.

5.2.3 Comparison with Stage

Vaughan (2008) conducted Stage's performance evaluation on an Apple MacBook Pro, with a 2.33 GHz Intel Core 2 Duo processor and 2 GB RAM. For the evaluation, each core in the machines I utilized offers comparable features: 2 GHz speed, 1 GB RAM per thread when $P = 16$.[3]

Stage can simulate approximately 10^3 robots in real-time, according to the results of experiments run without graphics, in a large environment with obstacles and with simple wheeled robots (Vaughan, 2008). These results were obtained with Stage version 3, whose architecture is single-threaded and physics is limited to 2D-kinematics equations. Under similar circumstances, when ARGoS is executed without threads and with a single 2D-dynamics physics engine, 10^3 robots are simulated in 24% of the corresponding real-world experiment time.

5.3 Results with Other Physics Engines

5.3.1 2D-Kinematics Engine

One of the engines available in ARGoS is a custom-made 2D-kinematics engine. This engine is designed to support simple navigation-based experiments involving a low number of robots (up to a few hundred). No effort was made to optimize the code for scalability. For instance, in this engine the computational complexity of collision checking is $O(N^2)$. Due to its extreme simplicity, the 2D-kinematics engine is a good test to prove the advantages of running multiple engines in parallel.

The left side of Figure 5.5 shows the average wall clock time and speedup of the benchmark experiment when a custom 2D-kinematics engine is employed. All the experiments summarized in the plot were performed with $P = 16$ threads, and with space partitioning A_1 to A_{16}. Results indicate that, when the space is partitioned among 16 kinematics

[3] With $N = 10^5$ robots, ARGoS used about 800 MB of RAM.

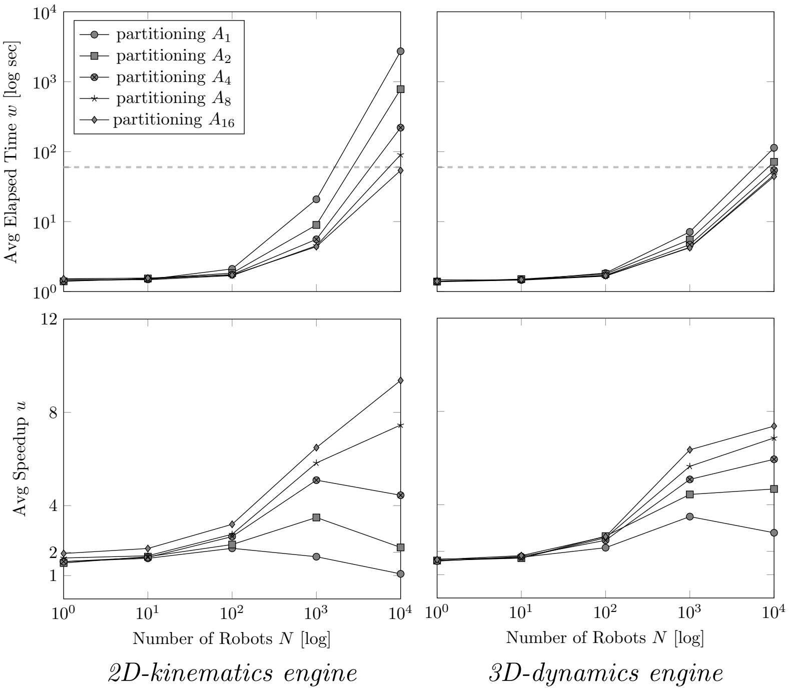


Figure 5.5: Average wall clock time and speedup for experiments with 2D-kinematics engines and 3D-dynamics engines. Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, P, A_{16} \rangle$. Each experiment simulates $T = 60\,\text{s}$. In the upper plots, points under the dashed line signify that the simulations were faster than the corresponding real-world experiment time; above it, they were slower. Standard deviation is omitted because its value is so small that it would not be visible on the graph.

engines, ARGoS is able to simulate $N = 10^4$ robots in approximately real-experiment time. Thus, even though the kinematics engine was not designed to scale, by using multiple instances of this engine it is possible to enhance performance to simulate thousands of robots.

5.3.2 3D-Dynamics Engine

The most capable physics engine in ARGoS is a 3D-dynamics engine based on the ODE library. As explained in Section 2.2.2, this engine is used by Webots and Gazebo, two very successful robot simulators.

In the right side of Figure 5.5, I report the results of the benchmark experiments conducted with this engine. Analogously to the experiments with the 2D-kinematics engine, these experiments were run with $P = 16$ threads and with space partitionings A_1 to A_{16}. In the wall clock time graph, the measured timings are very close to each other, although the best result is obtained when $E = 16$ engines are used. The lowest wall clock time obtained for $N = 10^4$ is approximately T, so, once more, ARGoS can simulate $N = 10^4$ robots in real-experiment time even with an accurate 3D-dynamics engine.

Summary

In this chapter, I reported the experimental activities I conducted to assess the efficiency of ARGoS. The analysis is based on a benchmark experiment derived from the performance assessment of the Stage simulator (Vaughan, 2008). The benchmark consists of a swarm of robots that diffuse in a large structured environment. I identified three main factors in the run-time performance of ARGoS: the number of robots involved, the number of threads employed, and the number of regions in which the physical space is partitioned. I constructed each experimental setup selecting a specific value for each factor.

I performed a first set of experiments using the 2D-dynamics engine of ARGoS. The experiments show that the benefit of using more threads and partitioning the space into more engines increases with the number of robots. Thus, for little swarms, it is advisable to employ one physics engine and configure ARGoS to work with one thread; for large swarms, however, parallelism and space partitioning are paramount to increase efficiency. The most remarkable result obtained in this set of experiments is that ARGoS is capable of simulating 10,000 robots in 60% of real-time.

I ran further sets of experiments with the other two physics engines available in ARGoS, the 3D-dynamics and the 2D-kinematics, under the same experimental conditions. The results confirm the findings of the first experiment set. Additionally, even when a physics engine has not been designed for efficiency, the results show that ARGoS is capable of ensuring high levels of efficiency.

Chapter 6

Validation

In this chapter, I present three experiments that showcase the correspondence of the results obtained in simulation with respect to those obtained in real-world experiments. These experiments have been conducted as part of research projects in which I cooperated. My contribution to these projects was designing and implementing the software infrastructure, both for the simulations and for the real robots. Participating to these experiments provided me with precious feedback on the design of ARGoS, allowing me to refine and improve it.

The ultimate aim of validation is to confirm that predictions made by simulation are matched by real-world experiments in comparable conditions. For this reason, in the experiments presented in this chapter, I concentrate on the correspondence between simulated and real-world swarm behaviors, rather than on the accuracy of each individual model employed in ARGoS. In other words, I consider the models accurate enough if the swarm behavior they predict matches the findings in the real world.

The first experiment, described in Section 6.1, involves a swarm of robots performing flocking in different situations. These experiments suggest that the standard models of ARGoS are capable of capturing the essential motion and communication features of the robots employed.

The second experiment, illustrated in Section 6.2, consists in a scenario in which the robots must navigate an unknown environment exchanging information. The behavior of the robots in simulation displays an interesting emergent property that is confirmed by real-world experimentation.

The third experiment, discussed in Section 6.3.1, shows a case in which the available models are not accurate enough to ensure an acceptable correspondence between simulation and real robots. Thus, I describe how ARGoS was extended with new device models that provide the necessary level of accuracy.

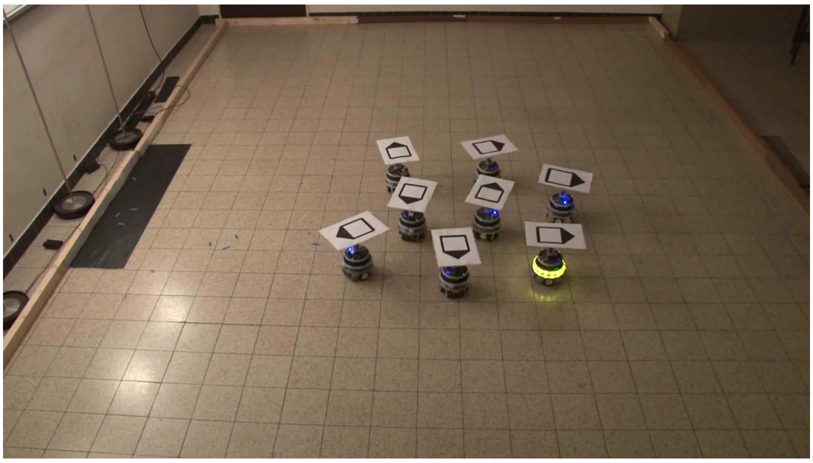


Figure 6.1: A screen-shot from the flocking validation experiments of Section 6.1. The robot with the yellow LEDs lit is the only one aware of the target direction.

6.1 Flocking

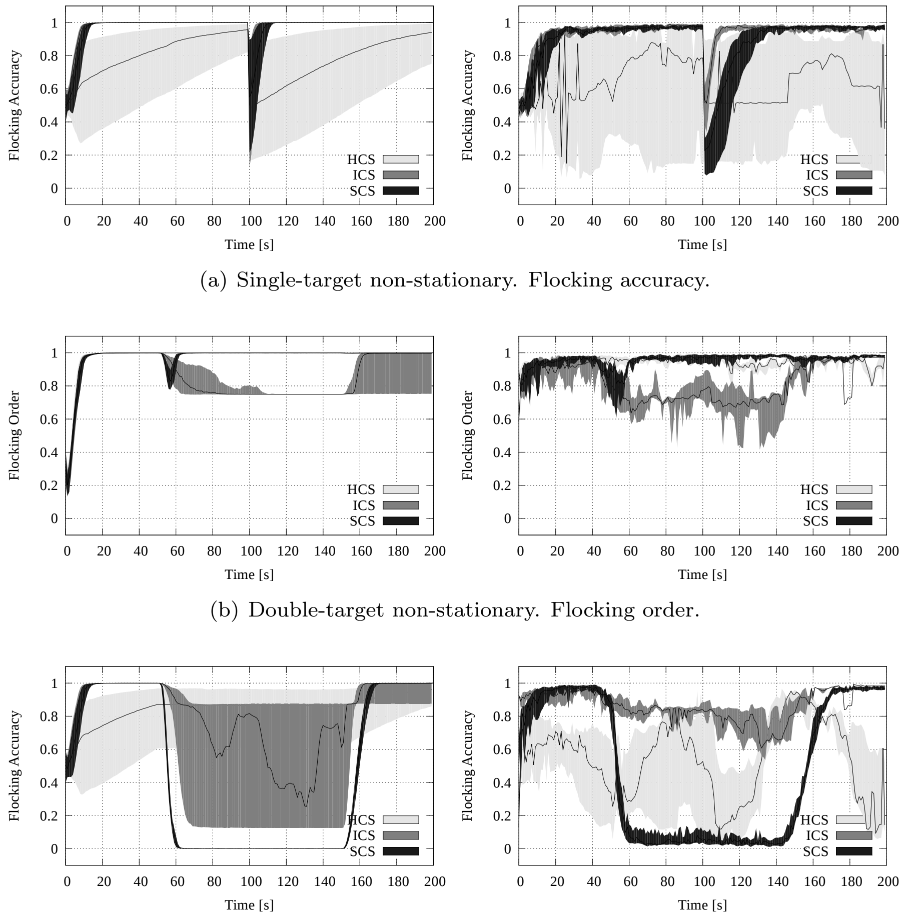
Often, in swarm robotics scenarios, groups of robots are required to navigate in cohesive groups towards a target location. In the literature, this behavior is called *flocking*. In this section, I describe and compare three flocking algorithms (Ferrante et al., 2013). As it will be illustrated, results indicate that the standard device models available in ARGoS are sufficiently accurate to characterize the behavior of these algorithms.

6.1.1 Experimental Setup

The experiments take place in three scenarios. All involve a group of 8 foot-bots deployed in an arena in which a light source is placed far from the robots. This light is used by the robots as common environmental cue to build a shared reference frame.

In scenario 1, hereinafter *single-target stationary*, a target location is placed at one end of the arena, far from the light and the robots. One robot, chosen at random before the beginning of the experiment, can sense the position of the target location. This is the *informed* robot. The other robots are *non-informed*, i.e., they have no direct information about the position of the target location (see also Figure 6.1).

Scenario 2, hereinafter *single-target non-stationary*, begins like scenario 1. However, midway through the experiment, the target location is moved to the opposite end of the arena with respect to its current position. In addition, the currently informed robot is demoted to non-informed and a different robot is selected at random to replace it.



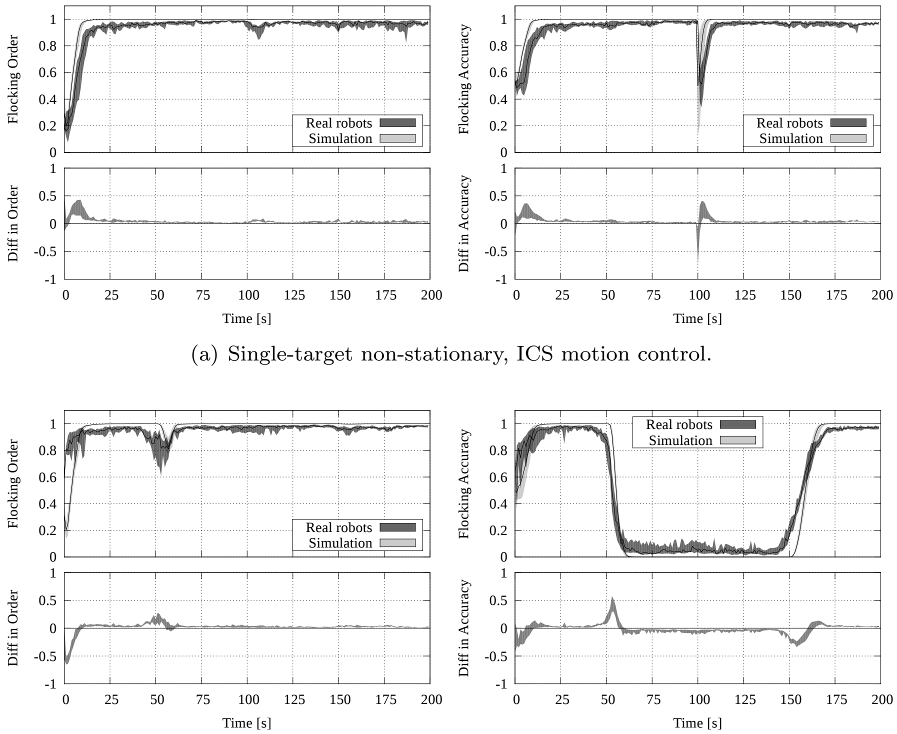
(a) Single-target non-stationary. Flocking accuracy.

(b) Double-target non-stationary. Flocking order.

(c) Double-target non-stationary. Flocking accuracy.

Figure 6.2: Results of the flocking validation experiments of Section 6.1. The plots on the left show the result obtained in simulation, while those on the right report the results in reality. The colored areas span the result distribution from the first to the third quartile. Each plots reports the data obtained with three different motion control strategies: HCS, ICS, and SCS. Refer to Section 6.1.2 for more details on the motion control strategies.

Scenario 3, *double-target non-stationary*, proceeds like scenario 2. Initially, a single target location is present and a single robot (*R1*) is informed. Midway through the experiment a new target location is added, and a new robot (*R2*) is promoted to the informed state. However, differently from scenario 2, the first target location is not removed. Thus, two informed robots with opposite target locations coexist in the swarm. The new target location is assigned the highest priority, so the swarm must abandon the path to the first target location and switch to the new one as quickly as possible. Finally, after a certain period



(a) Single-target non-stationary, ICS motion control.

(b) Double-target non-stationary, SCS motion control.

Figure 6.3: Results of the validation experiments of Section 6.1. The plots on the left show a comparison of the flocking order obtained in simulation and with real robots. The plot on the left show a comparison of flocking accuracy. Each plot is composed of two stacked elements: the top element reports the data, the second element reports the results of a Wilcoxon signed-ranked test on the difference between simulated and real-world data.

of time, the new location is removed and *R2* is reset to non-informed state. Thus, the swarm is required to resume navigation towards the first target location.

6.1.2 The Robot Behaviors

The aim of these experiments is to compare the correspondence of the dynamics of three algorithms in simulation and in the real world. The algorithms share the same communication modalities and the same structure. The robots communicate locally and wirelessly through the range and bearing system, and move according to vector $\mathbf{f} = \alpha\mathbf{p} + \beta\mathbf{g}_j + \gamma\mathbf{h}$. Vector $\mathbf{p}$ accounts for attraction and repulsion to nearby robots. It is calculated as described by Hettiarachchi and Spears (2009). Vector $\mathbf{g}_j$ encodes the goal direction, i.e., the direction to the target location. The value of j indicates the information available to a robot. When $j = 0$, the robot is non-informed, thus $\mathbf{g}_0 = \mathbf{0}$. When $j > 0$, the robot is informed about the j-th target location. Vector $\mathbf{h}$ is used to let the robot swarm align towards a common direction. The calculation of this vector assumes that *(i)* the robots can measure their attitude with respect to the common reference frame defined by the light and that *(ii)* the robots communicate attitude information to each other. It is important to note that sensor noise strongly affects the attitude measurement of a robot, and, thus, the behavior of the algorithms. In fact, the attitude is derived from the position of the light in the environment, which is perceived by the robots through noisy light sensors. Each robot calculates vector $\mathbf{h}$ as the average of the attitude information received by nearby robots. The three algorithms being compared differ in the data communicated among the robots.

In the first algorithm, Heading Communication Strategy (HCS) (Turgut et al., 2008), the robots communicate their current attitude with respect to the common reference frame.

In the second, Information-Aware Communication Strategy (ICS) (Ferrante et al., 2010), the informed robots broadcast the angle of the goal direction and the non-informed robots broadcast the angle of the average attitude direction received at the previous control step.

In the third algorithm, Self-Adaptive Communication Strategy (SCS) (Ferrante et al., 2013), the data sent by a robot R depends on *(i)* whether R is informed or not, and *(ii)* on the level of confidence in the information to be sent. In brief, if R is informed and its confidence is high, it broadcasts the direction to the target area. On the other hand, if R is non-informed and its confidence is low, R broadcasts the average of the directions received from the other robots at the previous control step. For medium levels of confidence, R sends a linear combination of the goal direction (if R is informed) and the average of the other directions received. An important aspect to note for validation purposes is that, in all three algorithms, sensor noise strongly affects the calculation of $\mathbf{h}$.

6.1.3 Performance Measures

To compare the results obtained in simulation and in reality, we employed two performance measures: *flocking order* and *flocking accuracy*.

Flocking order is calculated as the sum of the attitude vectors of the robots in the swarm. The length of each robot's attitude vector is set to 1, so the length of the resulting vector sum is in the range $[0 : 1]$. When flocking order is close to 1, the robots are aligned. Lower values correspond to robots pointing in different directions.

Flocking accuracy measures the difference between the sum of the attitude vectors and the target direction. Values close to 1 correspond to high accuracy. Exceptionally, in the second phase of the double-target non-stationary scenario, since the target direction is opposite to the direction of the other phases, values close to 0 correspond to high accuracy.

6.1.4 Results

We ran 100 experiments in simulation and 10 in reality for each scenario. Figure 6.2 shows a selection of results obtained in simulation and reality in scenario 2 and 3 with the three algorithms. Qualitatively, the performance of the algorithms in simulation and with real robots is very similar.

To further analyze the correspondence of results in simulation and in reality on a quantitative level, we performed a set of Wilcoxon signed-rank tests on the difference between them. More precisely, for each time step of an experiment (in a specific scenario and with a specific algorithm), we have 100 measures in simulation and 10 measures in reality for flocking order, and the same for flocking accuracy. We executed a Wilcoxon signed-rank test on the difference between the two data sets for each performance measure. The result is illustrated in Figure 6.3. The top plot of each block reports the raw measure, while the bottom plot shows the minimum and maximum bounds on their difference. As it can be seen from these results, the correspondence of the behavior of the algorithms is very high. These results support the statement that predictions based on simulations in ARGoS about the algorithms under study hold true for experiments with real robots.

It is common for algorithms developed in simulation to require modifications to work on real robots. Modifications typically span from parameter tuning to code rewriting. Remarkably, in the presented experiments, such high level of accuracy was achieved without any modification.

6.2 Cooperative Navigation

In this section, I discuss the results obtained with an algorithm for cooperative navigation in an unknown environment. The algorithm allows a robot to reach a target location in the arena under the guidance of other kin robots. The experimental scenario is reported in Figure 6.4. The left side of the figure shows the real robot arena, while the right side depicts the equivalent simulated arena.

6.2.1 The Robot Behavior

The navigation algorithm is based on line-of-sight wireless communication — two robots can communicate only if there is no obstacle between them. In addition, upon receipt of a message, a robot receives not only

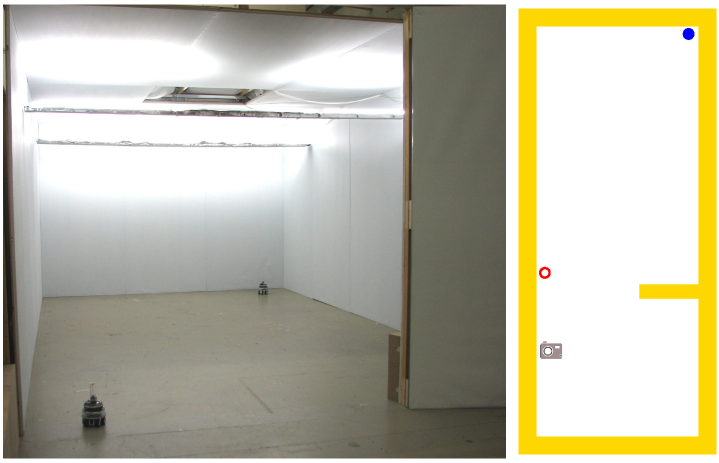


Figure 6.4: Setup of the validation experiments of Section 6.2. The picture on the left shows the real arena in which the experiments were conducted. The target locations the robots must visit are marked by dedicated robots. The diagram on the right depicts the equivalent simulated arena. The empty and filled circles represent the target positions, and the camera marks the location in which the picture on the left was taken.

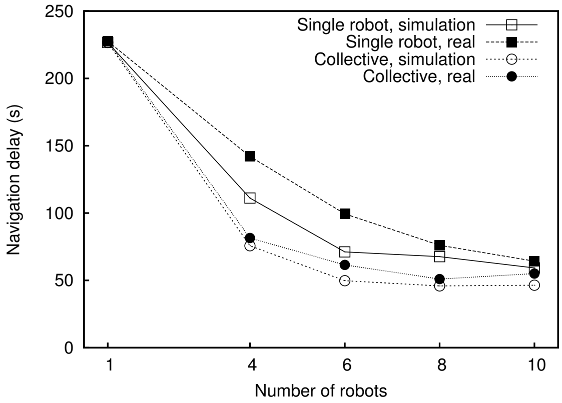


Figure 6.5: Results of the validation experiments of Section 6.2. The navigation delay is the time necessary for a robot to reach the target location. Here, the graph shows its average.

the payload of the message, but also the location (angle and distance) from which the message was sent. On the foot-bot, the range-and-bearing communication device provides these communication features.

The robots exchange navigation information through the wireless network, in order to guide the searching robot to the location of the target. In practice, each robot maintains a table containing information about known target robots, including estimated distance and age of each piece of information. Each robot periodically broadcasts the content of its table to the neighbors. Upon receipt of information from a neighbor, a robot updates its table accordingly. In addition, the distances are updated using odometry information. The searching robot traverses the network, monitoring at the same time the information broadcast by the guiding robots. The searching robot follows the highest quality information (with respect to age and distance to the target). For more details about this algorithm see (Ducatelle et al., 2013).

6.2.2 Results

The algorithm was tested in two scenarios. In both scenarios, two target locations are marked by static robots. In the first, a single robot must go back and forth between the two locations, while the other robots perform random walk and provide guidance. In the second scenario, the entire swarm is engaged in navigation between the two locations. This algorithm is a good validation test because, despite the simplicity of its implementation, it shows rich dynamics.

In the first scenario, the searching robot follows navigation information to reach the target, following paths that become shorter when the number of guiding robots increases. In the second scenario, the entire swarm spontaneously self-organizes into a spinning loop structure that includes the two target locations. Also in this case, navigation efficiency improves as the number of moving robots increases. Both simulated and real experiments showed the same qualitative dynamics in the two scenarios.

To better quantify the correspondence between simulated and real experiment, as performance measure we defined the time needed by a robot to go from its initial location to the target area (called *navigation delay*). We tested the algorithms in both scenarios with numbers of moving robots ranging from 1 to 10. In the experiments with real robots, we ran one 30-minute-long experiment for each number of robot, to allow the robots to visit the two target locations several times. The graph in Figure 6.5 reports the average navigation delay for each experimental setup. Results confirm a satisfactory correspondence between simulated and real experiment for both scenarios.

6.3 Task Partitioning in Cooperative Foraging

In this section, I describe an experiment in which the standard foot-bot model provided by ARGoS proves to be not sufficiently accurate. I discuss how ARGoS can be extended to include a better robot model, thus providing the necessary accuracy.

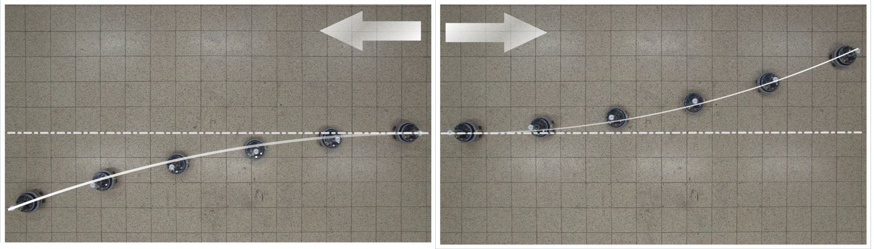


Figure 6.6: When the same speed is applied to the foot-bot treels, the robot does not cover a straight line, due to an asymmetry in the construction of the treel motors.

The experiment involves cooperative foraging by a swarm of foot-bots. The robots' behavior is designed to face non-trivial, real-world issues that significantly impact both the exploration of the environment and the transportation of target objects.

6.3.1 Experimental Setup

A swarm of 6 foot-bots is deployed in an area of the environment, called the *nest*. The robots must bring some objects to the nest. These objects are placed 4 m far from the nest, in a location, the *source*, which is unknown to the robots. Thus, the robots must first explore the environment to discover the source, and then proceed with object transportation. At any moment, each robot is capable of transporting one object at most. The robots continue the exploration/transportation routine until a certain time limit is reached.

The primary issue in this scenario is navigation from the nest to the source and vice-versa. The literature abounds with methods to navigate between two locations. To maintain minimal requirements on the robots, in this scenario, we assume that a robot can only use dead-reckoning, that is, it can estimate its position with respect to a certain location from the integration of odometry information. This method, however, is very sensitive to sensor noise. In fact, as a robot navigates, the integration of noisy odometry information causes an accumulation of the estimation error on the position of the target location. On the real foot-bot, over time the amount of error becomes so high that the robot must discard odometry information and return to exploration. This error is mainly caused by an asymmetry in the construction of the wheel motors. As shown in Figure 6.6, if both wheels of a foot-bot are set at the same forward speed, the robot's trajectory slants to the left. Analogously, when moving backwards, the trajectory is slanted to the right. Thus, a robot that moves back and forth between two points draws an S-shaped trajectory instead of a straight one. The magnitude of the slant

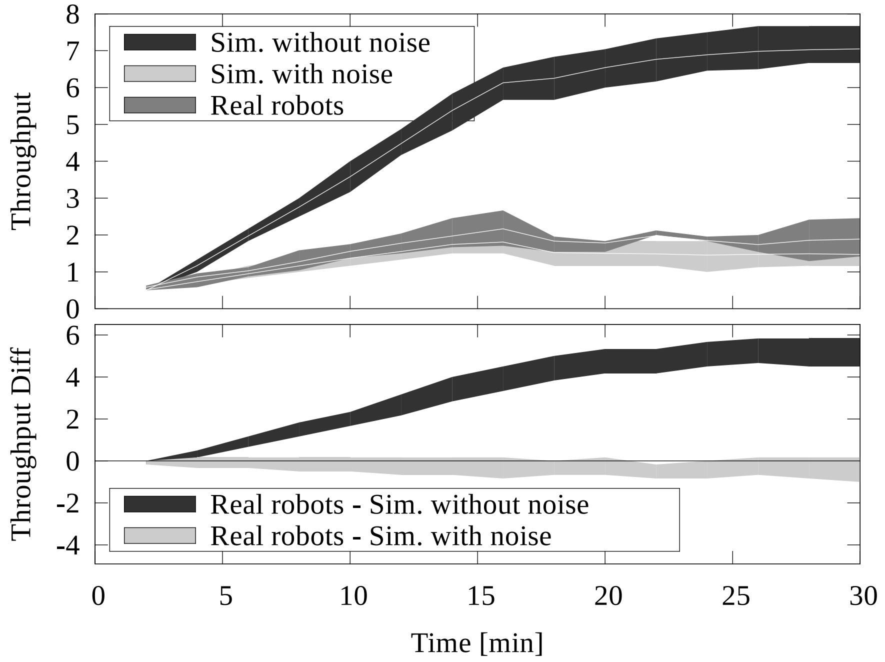


Figure 6.7: The throughput of object transportation in simulation with and without noise and on real robots. The top plot shows the interquartile range (Q25–Q75) of the raw data; the bottom plot reports the results of a Wilcoxon signed-rank test on the difference between simulation without noise and real robots, and simulation with noise and real robots.

is different across each trip and it is undetectable by the on-board motor sensors.

6.3.2 The Robot Behavior

The behavior of the robots is based on the idea that the estimation error increases with the distance covered by a robot. Thus, to limit the error, it is enough to limit the range of motion of the robots. In practice, the task of moving an object from the source to the nest is partitioned into several sub-tasks consisting of moving the object for a short distance. Each step of this process is performed by a different robot. For more details about this behavior, see (Pini et al., 2013).

6.3.3 Dead-reckoning Model

The standard implementation of the wheel motors in ARGoS does not include noise. This is a common choice in many simulators (e.g., Stage, Gazebo, Webots) because, as demonstrated by the experiments in Sec-

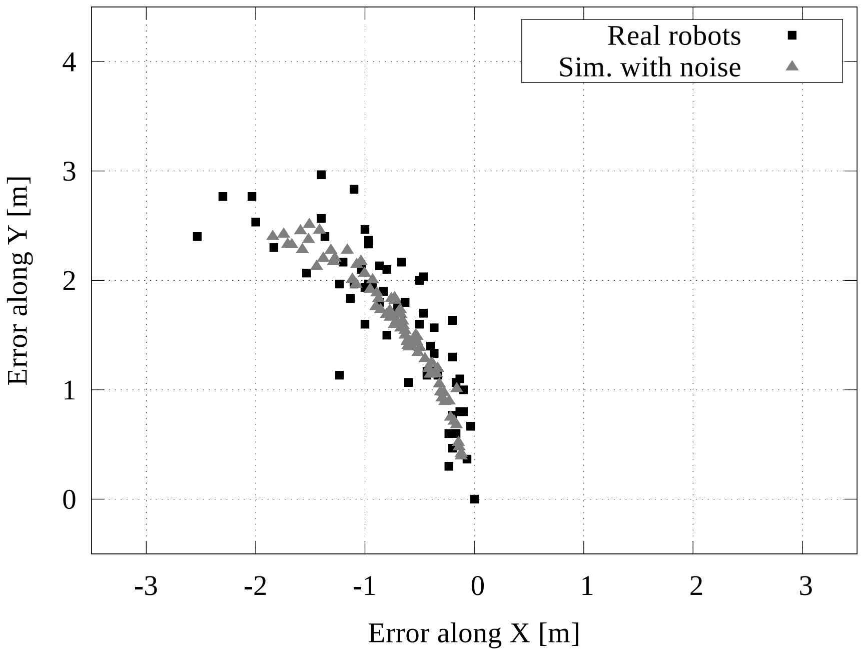


Figure 6.8: Positioning error of the foot-bots with respect to their target location. I report both the data sampled from real robot experiments and from the dead-reckoning model described in Section 6.3.3.

tions 6.1 and 6.2, in the vast majority of experiments the impact of this kind of noise is negligible.

However, in the experiment under study, such noise plays a fundamental role. In Figure 6.7 I report the throughput of objects brought to the nest. The throughput is calculated as the number of objects that reached the nest in the previous fifteen minutes. Throughput samples are collected every two minutes. At the top of Figure 6.7 I report the raw throughput data in three cases: simulation without noise, simulation with noise and real robots. As shown, when no noise is added to the actuators, the results in simulation do not reflect the data collected with real robots. A quantitative measure of the correspondence, or lack thereof, between the two data sets is reported at the bottom of the same figure. The bottom plot shows the 95% confidence interval on the difference between the data sets computed through a Wilcoxon signed-rank test. The result of this test on the difference between the data in simulation without noise and on real robot demonstrates that, for this experiment, the predictions of the standard ARGoS model is too optimistic. Thus, we constructed a noise model and added a new actuator with such model into ARGoS.

To construct the noise model, we analyzed a set of videos of the motion of real foot-bots during navigation to the source. We collected a set of 61 positions, derived the noise model, and implemented a new ARGoS module for the actuator. To reproduce the slanted motion, the actuated wheel speed is obtained by summing a random term to the desired wheel speed set by the robot. If the motion is forwards (positive speed value), the random term is summed to the desired right wheel speed; if the motion is backward (negative speed value), the random term is subtracted from the desired left wheel speed. The random term is obtained by multiplying the desired speed by a term μ, taken at random from a Rayleigh distribution ($\sigma = 0.00134$). The value of μ is chosen at random at the beginning of the experiment, and subsequently changed every time a robot grips an object. Figure 6.8 reports the samples from real robot experiments and the data obtained with the described model.

This model is very simple and does not include an explicit representation of the wheel motors. Despite its simplicity, the results illustrated in Figure 6.7 confirm that the throughput of the robots in simulation matches the throughput in real robot experiments.

6.3.4 Implementation in ARGoS

As discussed in Section 3.2, ARGoS offers two approaches to include new features or better models.

The first approach involves creating a new module implementation. For instance, in the experiment under consideration, the improved dead-reckoning model can be included in a new implementation of the wheel encoder sensor of the foot-bot. Alternatively, the experimenter can code a suitable loop function hook (see Section 3.2.7).

For sensors and actuators, the first approach is usually preferable when the added features cover relatively general use cases. In contrast, if the added feature is considered experiment-specific or of little general interest, a loop function hook is a wiser choice.

For the implementation of the noise model above discussed, we selected the first approach. On the other hand, the loop functions proved to be necessary to implement two aspects of the experiment: *(i)* the logic whereby a target object dropped in the nest is moved back to the source area, and *(ii)* the collection of data reported in Figure 6.7 and 6.8.

6.4 Discussion

The experiments presented in this chapter confirm that the basic models in ARGoS cover a wide range of applications, and are capable of predicting swarm-level behaviors. In addition, when the models offered by ARGoS do not provide satisfactory accuracy, it is easy to define and utilize a new model that provides the desired accuracy.

An interesting aspect about the models employed in the experiments in Sections 6.1 and 6.2 is that capturing complex real-world behavior does not require the use of computationally expensive models. In other words, the dynamics of swarm behavior appears to be sufficiently robust to cope with low accuracy in the basic models, provided that the latter at least capture the essential features of the system. This aspect is not completely surprising, because it is well-known that swarm robotics systems are capable of providing reasonable levels of performance even in presence of significant noise (Hoff et al., 2011). Low model accuracy could be considered as a form of noise over the dynamics of the swarm system. In any case, I believe that the question of 'minimal' modeling of swarm systems is open and worthy of further investigation.

Summary

In this chapter, I presented three sets of experiments meant to validate the results obtained with ARGoS in simulation against results obtained in real-world experiments.

The first set of experiments involved a problem of coordinated motion (often referred to as 'flocking'). The experiments confirmed that the default motion and communication models in ARGoS are sufficient to predict the behavior of a swarm of flocking robots under various conditions.

The second set of experiments consisted in a scenario of cooperative navigation in an unknown environment. In a preliminary experiment, a swarm of robots guides a robot between two points in the environment. In a second experiment, the entire swarm is engaged in mutual guidance. For both scenarios the predictions of ARGoS were matched by real-world experimentation. In particular, in the swarm-wide experiment, simulations correctly predicted an emergent behavior whereby robots form a spinning loop between the two target points in the environment.

The third set of experiments showed a case in which the default models of ARGoS were not sufficient to capture the essential features of the robot dynamics. I discussed the limits of the models and showed how new models have been derived and inserted into ARGoS, resulting in a simulation that matched the real-world dynamics.

Chapter 7

Team Recruitment and Delivery in a Heterogeneous Swarm

In this chapter, I present a scientific work that demonstrates the capabilities of ARGoS. The work consists in a novel, distributed approach to form multiple groups of mobile robots with precise control on the group size. This approach takes inspiration from the aggregation behavior of cockroaches under shelters.

In nature, shelters are passive landmarks beneath which cockroaches stop. In this system, mobile robots play the roles of cockroaches. By making shelters into active components with basic communication and sensing capabilities, I obtained a system that achieves the formation of multiple groups of robots in parallel.

I assessed the properties of the system through mathematical models and experiments based on physical simulations. In particular, I showcase the performance of the system in a challenging scenario, in which hundreds of robots must be organized into dozens of groups.

7.1 Introduction

One of the main benefits of swarm robotics systems is their potential for parallelism. To achieve parallelism in real-world scenarios, it is important to be able to split the swarm into appropriately sized groups for different concurrent tasks.

Imagine a swarm of robots that must be deployed to monitor the spread of an environmental hazard. Different hazard areas of various sizes will need correspondingly sized groups of robots, and the hazard sites may be spread far apart. As in any such real-world scenario, it is likely that there will not be enough robots to allocate the ideal number to each hazard site. In this chapter, I propose a distributed mechanism to solve this type of group formation problem, whereby large numbers of robots must be divided into multiple groups in parallel. When the number of available robots is sufficient, this system is capable of forming groups of different, pre-defined sizes. When the available robots are less than the sum of the desired sizes, the system distributes robots fairly across groups, ensuring that each group grows at the same rate.

Existing approaches to parallel group formation in multi-robot systems have limitations that render them inappropriate for this type of scenario. Decentralized task allocation and task partitioning approaches scale well, but they only work when the tasks are located close to each other and the density of robots is sufficiently high (Gerkey and Matarić, 2004). Centralized approaches, on the other hand, can efficiently divide a population of robots into specific group sizes matching different tasks, but do not scale to large swarms of robots due to high communication overheads (Dias et al., 2006). The only decentralized approaches that have tackled the formation of specific group sizes in swarms of robots have restricted themselves to forming or counting a single group of robots at a time (Melhuish et al., 1999; Brambilla et al., 2009).

This work is inspired by the aggregation behavior of cockroaches under shelters. The dynamics of this behavior are well understood, and predict different aggregated group sizes in an environment with different sized shelters. In (Amé et al., 2006) cockroaches are modeled as simple agents that walk randomly in an environment, and have a certain probability of stopping that increases with the presence of a shelter and with proximity to other stopped cockroaches. This model is a good fit for the parallel group size selection problem in robotics, as there is no communication required between agents. This approach does not, therefore, suffer from the scalability limits of centralized approaches. I use 'active shelters,' that can affect the probability of robots stopping underneath them within a certain communication range. By varying the stopping probabilities associated with different shelters, I reproduce the effect of different shelter sizes and create robotic groups of different required sizes.

In Section 7.2, I contextualize the work with respect to other related studies. In Section 7.3, I sketch the methodology by showing how I apply the cockroach model, and by introducing the properties the system must display. In Section 7.4, I illustrate the hardware necessary for the implementation of the system, and present the actual devices I used for this study. In Section 7.5, I introduce a basic version of the system. I model the system mathematically and analyze it compliance to the desired properties. Subsequently, I show how the system can be improved to satisfy better the desired properties. In Section 7.6, I show that the system displays scalability for increasing numbers of robots and shelters.

7.2 Related Work

In swarm robotics, existing approaches to the problem of splitting up a swarm of robots into groups revolve around distributed forms of task allocation (Gerkey and Matarić, 2004)—scalable, decentralized approaches with low or zero communication overheads. In these approaches, individual robots have a mechanism for testing the task and allocating themselves to different tasks or sub-tasks. Testing by robots can, for example, take the form of determining whether an object is moving in a group transport scenario (O'Grady et al., 2011), or testing the time taken to perform a task or sub-task (Pini et al., 2011). The performance (and/or convergence time) of these methods typically depends on the number of interactions among robots per time unit, which, in turn, depends on the density of robots in the environment. An ideal density range exists, in which these methods perform best. Below a certain density threshold, the robots interact too rarely for these methods to be practically applicable. Above a certain threshold, the robots spend significant time avoiding each other rather than performing their assigned tasks. In real-world scenarios, the available robots are often scarce, tasks may be located far apart from one another, and explicit group sizes may need to be allocated based on some external (human) assessment of the problem. Under these circumstances, task-allocation-based approaches may not be feasible.

Explicit group size formation has been demonstrated as feasible in the swarm robotics context, but only for a single group at a time. Melhuish et al. (1999) controlled group sizes in a swarm of abstract agents using a firefly-like synchronization mechanism. However, only one group was formed at a time and the physics of the agents was not taken into account. Brambilla et al. (2009) used physically embodied robots, but their system could only passively count the number of robots in a single existing group (rather than generating a group of a priori determined target size). Hsieh et al. (2008) have studied an abstraction of the problem I study, whereby robots must distribute across multiple sites in predefined ratios. Hsieh *et al.*'s work is based on statistical models of house hunting ants.

In multi-robot systems formed by a low number of individuals, market-based approaches provide a good solution to the problem of parallel group formation of explicit group sizes (Dias et al., 2006). Market-based approaches, however, have intrinsic scalability problems that render them less applicable to swarm robotics systems. In market-based approaches, all robots participate in an auction, and market forces determine the most appropriate allocation of robots to groups corresponding to particular tasks. However, the fact that every robot must participate in the auction process means that bandwidth and computation requirements increase quickly with the number of robots.

Our approach is inspired by the aggregation behavior of cockroaches (Rust et al., 1995). Cockroach behavior is accurately mimicked by Jeanson et al. (2004) model in which cockroaches stochastically switch from stopping to performing a random walk and back. The probability of stopping rather than random walking increases with the number of other nearby stopped cockroaches. A positive feedback mechanism then results in aggregation into groups. When multiple shelters of different sizes are available in the environment, cockroaches may form groups of different sizes that correspond to the sizes of shelter available (Amé et al., 2006).

Previous robotics studies have shown how cockroach behavior can be faithfully mimicked by a group of robots. Existing robotic implementations share key features of the cockroach model. In particular, the equilibrium distribution of agents depends passively on the initial configuration of the environment and on the static mapping of environmental conditions to stop/go probabilities (Garnier et al., 2005). None of the existing studies, however, have used the model to generate a priori desired group sizes, or explored how many such groups could be formed in parallel.

7.3 Methodology

The approach is inspired by the behavior of cockroaches under shelters. This behavior has been modeled using decentralized agent-based models, whereby each cockroach agent is attracted to the shelters and to other cockroaches (Amé et al., 2006; Jeanson et al., 2004). In these models, cockroaches wander randomly in the environment, and decide probabilistically when to stop under a given shelter, and when to leave a shelter based on the stopping probability parameters of the model. Amé et al. (2006)'s cockroach model predicts different aggregated group sizes in an environment with different sized shelters.

I mimic some of the basic dynamics of these cockroach models to achieve parallel group formation. In the system, mobile ground-based robots play the role of cockroaches, and simple ceiling-based devices play the role of shelters. Instead of using shelters of different sizes, however, I make the shelters *active*, in the sense that they can calculate the robots' stop/go probabilities and communicate them to the mobile robots. In practice, each active shelter continuously transmits stop/go probabilities to mobile robots in a limited radius. I refer to this local communication radius as the *communication range* of an active shelter.

Analogously to cockroaches, robots wander randomly in the environment. Occasionally, a robot encounters a shelter and decides probabilistically whether to stop underneath it or not. The decision is based on the stop/go probabilities transmitted by that particular shelter. If a robot decides to stop, it becomes part of the group associated with that shelter. Robots under a shelter may also probabilistically decide to leave the shelter (thereby leaving the associated group). By assigning different probabilities to different shelters, the system can form groups of different a priori determined sizes in parallel.

The system has a stochastic nature. The two main sources of randomness are robot motion, and the probabilistic decisions on joining/leaving a group. Robot motion influences the discovery of shelters by robots, as well as the formation and disbandment of the groups. The probabilistic group joining/leaving mechanism impacts the stability of the groups over time. To say meaningfully that it has formed groups of a particular size, therefore, the system must settle at some point into a state in which the size of individual groups can be recognized as stable. I refer to this property as *convergence*. I give a formal definition of convergence in Section 7.5.2.

For the system to display parallelism, the group formation mechanism must ensure that all shelters receive a fair share of robots. I refer to this property as *fair spreading*. In the approach, fair spreading occurs when the rate of growth of all group sizes is proportional to the desired sizes. In a system that displays fair spreading, convergence time does not depend on the number of groups to form. In addition, with fair spreading, a system offers equal treatment for all shelters when the number of robots available for group formation is lower than the number of robots desired in total.

In (O'Grady et al., 2009), I introduced and analyzed a first implementation of this system. The results indicate that the join and leave probabilities impact differently the dynamics of the system. The join probability mainly accounts for the rate of addition of robots into groups. The leaving probability accounts for the rate of robot loss of a group, and has dramatic effects on the dynamics of the system. Experimental results showed that setting the leave probability to a high value greatly improves spreading at the expense of group stability, thus making convergence detection by the active shelters difficult or impossible. A low value for the leave probability, on the other hand, results in high stability and unfair or non-existent spreading. The system, presented in Section 7.5, offers a solution to this trade-off, so as to ensure both easy convergence detection and fair spreading.

7.4 Hardware

I first discuss the minimum capabilities necessary for a hardware platform to use active shelters as a means of group formation. I then describe the actual devices employed in this work.

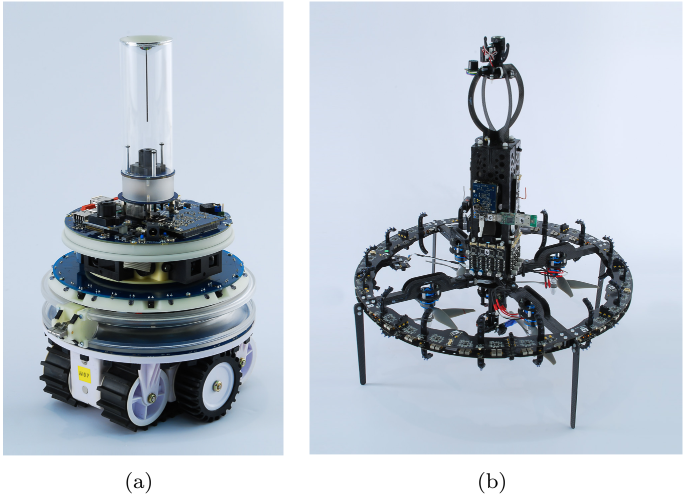


Figure 7.1: The robot platforms I simulated for the experiments in this study. (a) The foot-bot; (b) the eye-bot.

7.4.1 Minimum Generic Requirements

An active shelter is capable of monitoring the number of grouped robots over time, with coarse periodicity. In addition, it must be capable of broadcasting the join/leave probabilities to the robots in range. Finally, active shelters must be able to broadcast a 1-bit signal to nearby shelters.

A mobile robot must be capable of navigating the environment avoiding other robots. To coordinate the motion of robots joining/leaving a group as described in Section 7.5.2, each robot must convey 2-bit state information (joined-group/leaving-group/free) to nearby robots and shelters. A robot must detect the state of nearby robots and receive the join/leave probabilities from the closest shelter. Finally, a robot must be able to perceive nearby robots.

7.4.2 Devices Used in This Study

In the experiments, I employed the foot-bot (Bonani et al., 2010) as mobile robot (Figure 7.1 (a)). The foot-bot navigates the environment through a set of wheels and tracks called *treels*. It can convey its state (joined-group/leaving-group/free) through a colored LED ring that surrounds its body, and detect other robots' state and relative location through an omnidirectional camera.

To realize active shelters I used the eye-bot robot (Figure 7.1 (b)). The eye-bot (Roberts et al., 2007) is a quad-rotor robot able to attach to the ceiling. To detect mobile robots, the eye-bot is equipped with a pan-and-tilt camera.

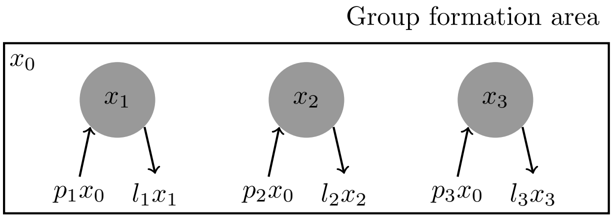


Figure 7.2: A schematic representation of the mathematical model described in Section 7.5.1 with three active shelters.

Eye-bot–to–foot-bot and eye-bot–to–eye-bot communication occurs through the range-and-bearing communication system (Roberts et al., 2009) present on all robots. This device allows the eye-bot to broadcast a message containing a 8-byte payload. The message can be received by nearby eye-bots and by foot-bots on the floor in a limited range.

The robots, as well as the range-and-bearing communication system, were developed in the Swarmanoid project (Dorigo et al., 2013).

7.5 Active Shelters

In this section, I present the system to achieve group formation control through active shelters. I analyze its properties and identify its limitations. In Section 7.5.1, I present a mathematical model that captures the most important aspects of the system. In Section 7.5.2, I present the complete implementation of the system and analyze it through targeted physics-based simulations.

7.5.1 Mathematical Model

The model.

In Figure 7.2, I illustrate the abstraction of the system upon which I base the mathematical model. The shelters are distributed in a rectangular area. The communication range of the shelters is displayed as a circular gray area. I assume that the communication ranges of a shelter do not overlap with those of the other shelters. For the purposes of this model, I neglect the actual motion of the robots across the environment. I assume that the robots perform a diffusion algorithm such as (Howard et al., 2002), so as to uniformly distribute in the environment. Thus, the probability c for a robot to be located within the communication range of a shelter is given by

$$c = \frac{\text{Area(shelter)}}{\text{Area(group formation area)}}.$$

A robot joins the group of shelter i with probability j_i. This probability is set by the active shelter depending on the desired group size. The

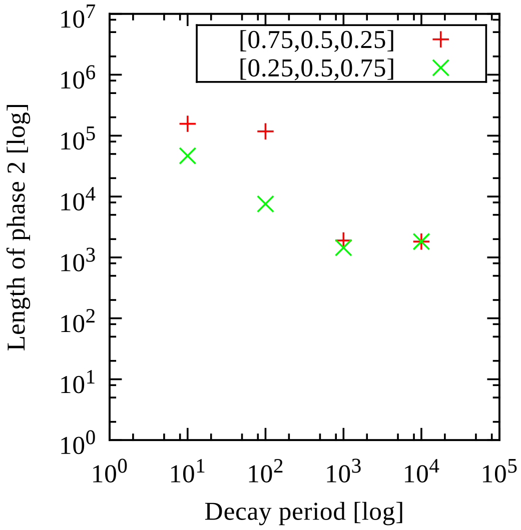


Figure 7.3: Length of phase 2 in the simulations of the mathematical model for different values of the decay period δ.

probability p_i for a robot to discover that shelter and join its group is given by $p_i = cj_i$. The probabilities p_i are assumed constant throughout the duration of a run.

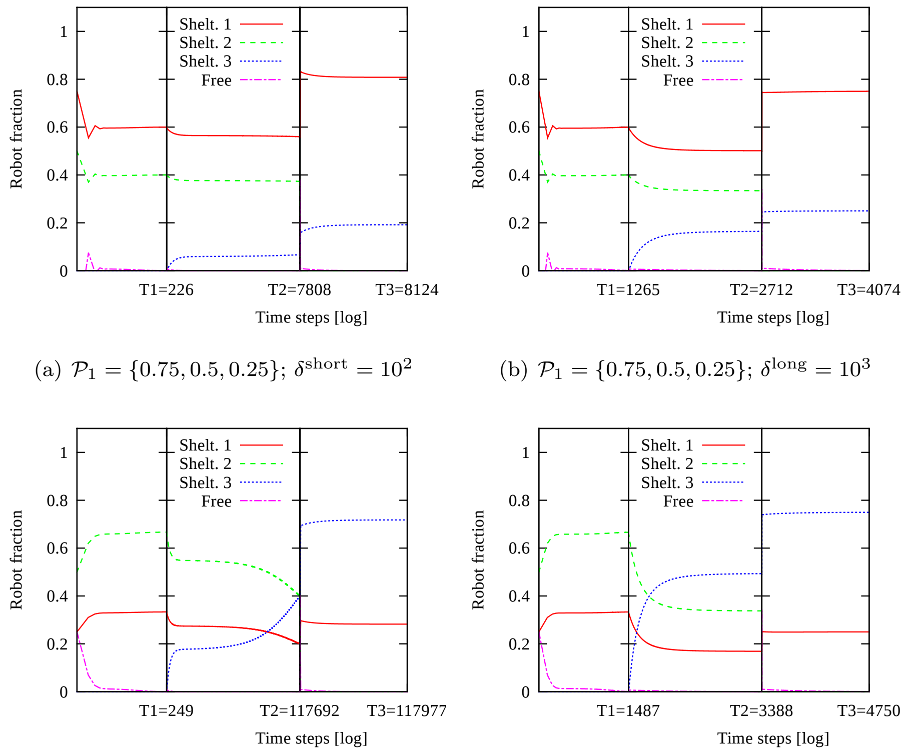
A robot that is part of group i has a probability to leave it, denoted by l_i. As discussed in Section 7.3, setting l_i to a fixed value causes a trade-off between fair spreading with difficult convergence on one side, and stable convergence with unfair spreading on the other (O'Grady et al., 2009). Thus, to reconcile convergence and spreading, the leave probability l_i must vary between a value that promotes fair spreading (hereinafter denoted by l^{hi}), and a value that ensures convergence (hereinafter denoted by l^{low}). At the beginning of the experiments, each shelter i is configured with a probability $l_i = l^{\text{hi}} = 10^{-2}$. I thus ensure a good spread of robots early on in the group formation process. Over a period of time, each shelter exponentially decreases the value of its l_i, until it reaches $l^{\text{low}} = 10^{-5}$, thus allowing the system to gradually settle into a stable state. More specifically, the values of the l_i's are set as follows:

$$l_i(t) = (l^{\text{hi}} - l^{\text{low}})e^{-\gamma t} + l^{\text{low}}, \tag{7.1}$$

in which γ is the decay constant. I define γ as a function of a parameter δ that represents the number of time steps required by the exponential to drop by 90%:

$$\gamma = \frac{\ln 10}{\delta} \qquad \Rightarrow \qquad e^{-\gamma\delta} = 0.1.$$

I denote the fraction of robots engaged in group $i > 0$ at time t as



(a) $\mathcal{P}_1 = \{0.75, 0.5, 0.25\}$; $\delta^{\text{short}} = 10^2$

(b) $\mathcal{P}_1 = \{0.75, 0.5, 0.25\}$; $\delta^{\text{long}} = 10^3$

(c) $\mathcal{P}_2 = \{0.25, 0.5, 0.75\}$; $\delta^{\text{short}} = 10^2$

(d) $\mathcal{P}_2 = \{0.25, 0.5, 0.75\}$; $\delta^{\text{long}} = 10^3$

Figure 7.4: Results with the mathematical model presented in Section 7.5.1. The experiments are composed of three phases. In phase 1, two shelters are active. In phase 2 (starting at time T_1), a third shelter is activated. In phase 3 (starting at time T_3), shelter 2 is deactivated. The experiment ends at time T_3. The length of each phase depends on the dynamics of the system. $\mathcal{P}_1$ and $\mathcal{P}_2$ account for the desired group size of each shelter. Parameter δ corresponds to the decay period for the probability to leave a shelter.

$x_i(t)$, and the free robots (i.e., not part of any group) as $x_0(t)$. Considering the probabilities p_i and l_i as rates of group joining and leaving, respectively, and denoting with n the number of shelters in the group formation area, yields the following model:

$$\begin{cases} x_0(t+1) = x_0(t) - \left(\displaystyle\sum_{i=1}^{n} p_i\right) x_0(t) + \displaystyle\sum_{i=1}^{n} l_i(t) x_i(t) \\ x_i(t+1) = x_i(t) + p_i x_0(t) - l_i(t) x_i(t) \qquad (\text{with } i \in [1, n]) \end{cases}$$

Model Behavior

To study the behavior of the model with respect to convergence and spreading, I set up a three-phase experiment. Initially, two shelters form groups of different sizes for a certain period T_1. At time $t = T_1$, a third shelter is activated and it starts recruiting robots. Subsequently, at time $t = T_2$, shelter 2 is disabled, thus freeing all the robots within its group. The simulation ends at time $t = T_3$.

The length of these three phases is not fixed, but depends on the dynamics of the system. To study convergence at each phase, I record the times T_1, T_2, and T_3 in which the system reaches convergence at each phase. Convergence is reached when all fractions (grouped and free robots) change by a sufficiently little quantity. More precisely, I declare convergence at time t^* if $\forall k \in [0, n]\; |x_k(t^*) - x_k(t^* - 1)| < 10^{-6}$.

Whenever a new shelter is activated or deactivated spreading is important to ensure fair resource distribution. The newly activated or deactivated shelter thus sends a signal to nearby shelters to notify them of the change to the system. The shelters react by forwarding this signal to their neighbors and resetting their leave probability value back to l^{hi}.

To study spreading, I ran simulations with two different settings for the values of the join probabilities: $\mathcal{P}_1 = \{0.75, 0.5, 0.25\}$ and $\mathcal{P}_2 = \{0.25, 0.5, 0.75\}$. With $\mathcal{P}_1$, shelters 1 and 2 initially must recruit most of the robots. Shelter 3 is the least demanding, so, upon its activation at time T_1, the system must redistribute only a few robots to shelter 3. Conversely, with $\mathcal{P}_2$, shelter 3 is the most demanding. Its activation at $t = T_1$ forces the system to redistribute most of the robots.

I ran several experiments with different values for parameter δ. The experiments revealed that the most problematic event in the setting is the addition of shelter 3 at the beginning of phase 2. As reported in Figure 7.3, the length of this phase decreases with the increase of the decay period in the range $[10^1, 10^3]$. For $\delta = 10^4$, the length of phase 2 slightly increases with respect to $\delta = 10^3$.

A sample of the dynamics of the system for different values of δ is reported in Figure 7.4. In these experiments, I tested $\delta = \delta^{\text{short}} = 10^2$ and $\delta = \delta^{\text{long}} = 10^3$ time steps. As the plots show, the length of the decay period of the leave probability has a strong impact on the behavior of the system and the values of T_1, T_2 and T_3. If the decay period is short ($\delta = \delta^{\text{short}}$), the value of the leaving probability drops quickly and the system behavior results in unfair spreading and very long convergence times. A long decay period ($\delta = \delta^{\text{long}}$), on the other hand, has a positive effect on both properties, because the robots have sufficient time to spread across the shelters.

7.5.2 Physically Simulated Robot

Implementation

I performed experiments using ARGoS to validate the predictions of the model in a physically realistic robotic system. A prominent aspect that affects the system performance is interference among mobile robots. In the system, interference occurs mainly under the active shelters. The mobile robots must organize in tight aggregates under the shelters, while permitting the flow of leaving and joining robots.

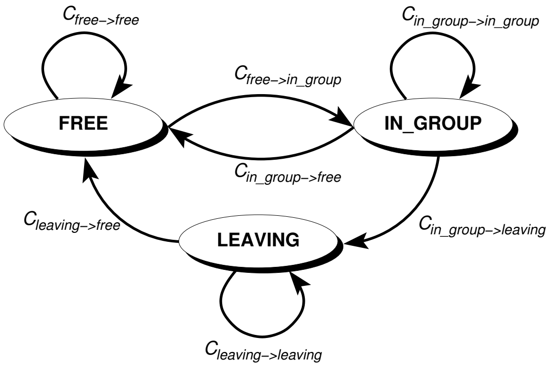
In the physics-based simulations, I assume that the robots are randomly distributed in a group formation area in which shelters are distributed. At any moment, a robot performing random walk across the environment can either be under a specific shelter or in a shelter-free area. The robot behavior is described by the simple state machine reported in Figure 7.5. The meaning of the states are as follows:

- state FREE: A robot does not belong to any group. The robot performs random walk with obstacle avoidance. This state is conveyed with LEDs lit up in green.
- state IN_GROUP: A robot is part of a group. This state is conveyed with LEDs lit up in red.
- state LEAVING: A robot is leaving a group to which it previously belonged. This state is conveyed with LEDs lit up in blue.

Join probability

When a robot in state FREE enters the communication range of shelter i, it transitions to state IN_GROUP with probability j_i. This probability is related to the target group size q_i associated to the shelter. I aim to find a simple relationship between j_i and q_i. The communication range of a shelter is limited in size and, thus, can house a maximum number of robots, which I denote with f. I assume that all the shelters have identical communication ranges, so f is a constant across the shelters known a priori. In addition, I assume that the target group sizes q_i cannot exceed f. Each shelter constantly monitors the number of robots currently engaged in its group. If the group size exceeds the target size, the shelter must stop recruiting robots. From these considerations, I derive the following simple definition for j_i:

$$j_i = \begin{cases} q_i/f & \text{if current size of group } i < q_i, \\ 0 & \text{otherwise.} \end{cases}$$


$C_{free\text{->}free}$
$C_{in\_group\text{->}in\_group}$
$C_{free\text{->}in\_group}$
FREE
IN_GROUP
$C_{in\_group\text{->}free}$
$C_{leaving\text{->}free}$
LEAVING
$C_{in\_group\text{->}leaving}$
$C_{leaving\text{->}leaving}$

State transition conditions	
$C_{free \rightarrow in\_group}$	`JustInRange()` $=$ *true* and `Rand()` $\leq j_i$
$C_{free \rightarrow free}$	`InRange()` $=$ *false* or (`JustInRange()` $=$ *true* and `Rand()` $> j_i$)
$C_{in\_group \rightarrow free}$	`InRange()` $=$ *false*
$C_{in\_group \rightarrow leaving}$	`InRange()` $=$ *true* and `Rand()` $\leq l_i$
$C_{in\_group \rightarrow in\_group}$	`InRange()` $=$ *true* and `Rand()` $> l_i$
$C_{leaving \rightarrow free}$	`InRange()` $=$ *false*
$C_{leaving \rightarrow leaving}$	`InRange()` $=$ *true*

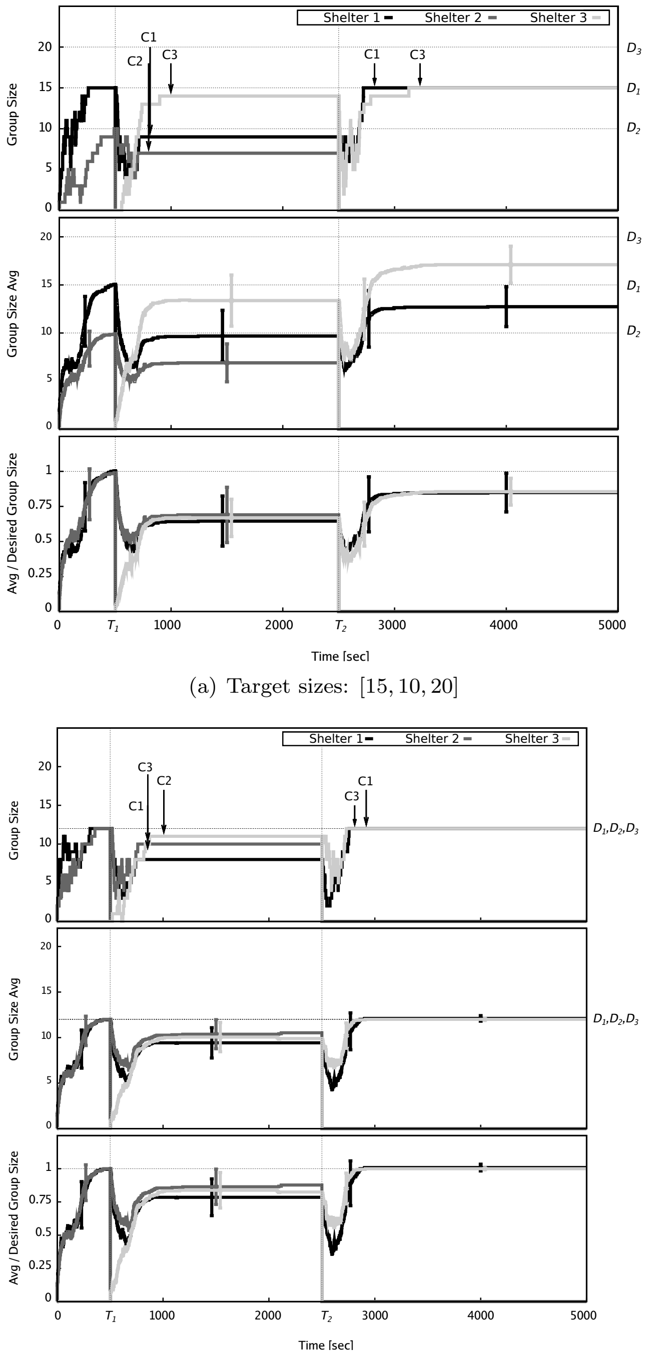
Figure 7.5: State transition logic for robots at each time step. `InRange()` and `JustInRange()` are functions returning *true* when the robot is within the communication range of a shelter, and has just entered it, respectively. `Rand()` is a function returning a random number in $\mathcal{U}(0,1)$. j_i is the join probability for shelter i, l_i is the leave probability. State transition conditions are represented be the symbol C and a subscript. For example, $C_{in\_group \rightarrow in\_group}$ represents the conditions under which an aggregated robot will stay aggregated in its group in a single time step.

Leave Probability

A robot in state IN_GROUP beneath shelter i transitions to state FREE with probability l_i. The probability l_i decays exponentially following (7.1).

Robots Joining and Leaving a Group

Physical interference among grouped robots may have severe effects on the system, especially when the density of robots under a shelter becomes high. Upon deciding to leave, a robot located in the center of a group needs a clean path out of it. However, if the area beneath a shelter is crowded, the formation of the exit path is likely to push some robots located at the border of the group out of the communication area, thus losing contact with the shelter. Thus, interference reduces stability, especially in large groups.



(a) Target sizes: [15, 10, 20]

(b) Target sizes: [12, 12, 12]

Figure 7.6: Results with physically simulated robots following the behavior explained in Section 7.5.2. The experiments are composed of three phases. In phase 1, two shelters are active. In phase 2 (starting at time T_1), a third shelter is activated. In phase 3 (starting at time T_2), shelter 2 is deactivated. The experiment ends at time T_3. The top plots shows a representative experimental sample in the pool of the 100 repetitions I ran. The middle plot reports the average system behavior. The bottom plot shows the ratio between the current and the desired group size.

To solve these issues, I consider each robot to be immersed in two virtual potential fields (Spears et al., 2004). The first field attracts robots that decide to be part of a group towards the center of such group, or repels those who decide to leave the group. The second field allows a robot that decides to leave the group to push its way out without disrupting the integrity of the group.

Convergence Detection

It is important for a shelter to be able to detect convergence. However, due to the probabilistic nature of the system, the group size displays continuous fluctuations. Thus, the notion of convergence to a target group size must be linked to these fluctuations. Intuitively, a shelter can declare convergence when the magnitude of the fluctuations remains for a certain period of time within a specific range.

In practice, each shelter monitors the fluctuations of the number of its aggregated robots over a period of time T_C. If the magnitude of the fluctuations stays within some tolerance boundaries for the entire period, the shelter considers the system to have converged. The tolerance boundary B is defined as a function of the leave probability l^{low}, the length of the monitoring period T_C and the size of the aggregate g_i at the beginning of the monitoring period:

$$B = \sqrt{T_C g_i l^{\mathrm{low}}(1 - l^{\mathrm{low}})}$$

B is derived by considering the changing number of aggregated robots under a shelter as a time series produced by a binomial distribution in which $p = l^{\mathrm{low}}$. B is calculated as the standard deviation of such a time series over the monitoring period T_C.

7.5.3 Experimental Evaluation

Experimental Setup

I ran experiments with a 3-phase setup analogous to what I presented for the mathematical model. The values of $T_1 = 50\,\mathrm{s}$, $T_2 = 250\,\mathrm{s}$, and $T_3 = 500\,\mathrm{s}$ were chosen to give the system sufficient time to reach convergence at each stage.

I distributed 30 robots in the group formation area. In the first phase, the target sizes of shelter 1 and 2 are always set so that their sum is less than 30. This first phase therefore allows us to test whether the system is capable of converging to the correct group sizes in the simple case where there are enough robots to satisfy all target sizes. The second phase, in which also shelter 3 is activated, tests the spreading property of the system in response to the activation of new shelters. In particular, I set shelter 3's target size to make the sum of the target sizes greater than the total number of robots in the group formation area. The third and final phase tests the spreading property of the system, this time in response to the release of robots by shelter 2. The convergence property of the system is tested in all three phases of the experiments.

For the experiments in this section, I tested two target size configurations: $[15, 10, 20]$ and $[12, 12, 12]$. In the first configuration, shelter 3's target size is greater than those of shelter 1 and shelter 2, thus requiring the system to redistribute robots. This corresponds to setting $\mathcal{P}_2$ of the experiments with the mathematical model. The second configuration was chosen to specifically assess the spreading property. For the decay period, I selected the value $\delta = \delta^{\text{long}} = 10^3$.

Results

The results are reported in Figure 7.6. The plots show that convergence is reached in both experimental settings, regardless of desired group sizes. The detection of convergence in phase 2, which is the most critical phase in the experiment, is achieved by all robots within slightly less than 10 simulated minutes from the beginning of the phase.

Regarding spreading, the bottom plots of Figures 7.6(a)-(b) show the average fulfilling percentage of the groups. In both experimental settings, regardless of the desired group size, these percentages stabilize on the same value in all phases.

7.6 Scalability Assessment

One of the key motivations to create a new group formation mechanism is to achieve scalability (see Section 7.2). This motivation is the driving force behind the choice of a decentralized cockroach-inspired model. In this section, I test the scalability properties of the approach. I conduct experiments with hundreds of mobile robots that must be divided in dozens of groups. Parallel group formation with these numbers would be difficult to achieve with existing group formation approaches.

7.6.1 Experimental Setup

The shelters are distributed in a $N \times N$ grid. I set the desired group size to 25 for all the shelters. The experimental setup is shown in Figure 7.7. Figure 7.7 (a) is a snapshot of the simulated group formation area. Figure 7.7 (b) is an abstract representation of the same group formation area in which the gray intensity of each square is proportional to the number of grouped robots. This representation allows for visual analysis of the sizes of the groups and their spatial distribution in the group formation area over time.

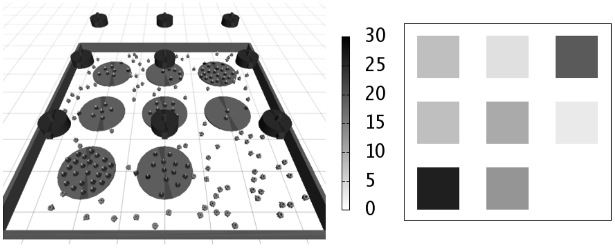


Figure 7.7: Snapshot from scalability experiments with physically simulated robots. Left: Simulation snapshot. Right: Abstracted representation of this simulation snapshot—the gray intensity level of each square is proportional to the recruited group size of the correspondingly positioned shelter (i.e., to the number of robots recruited by that shelter).

In all the experiments, I set the number of available mobile robots to $20N^2$. In this way, if fair spreading occurs, at convergence, a group of 20 robots should have been formed under each shelter.

7.6.2 Convergence

In this section, I present a set of experiments designed to test the convergence property of the system. I ran experiments with 16 and 25 shelters (320 and 500 mobile robots, respectively). The duration of each experiment was set to 750 s.

The results are reported in Figure 7.8. The top plots shows the dynamics of a sample run taken at random from the 80 experiment repetitions. I also display three snapshots at 250 s, 500 s and 750 s. After an initial period of instability, in which the group sizes grow with large fluctuations, the system converges in both experimental settings. The length of the decay period δ in these experiments was set to 350 s, which explains the duration of the fluctuating phase, about 400 s. At convergence, for both experimental settings, the distribution of the group sizes is centered around the target value of 20. In particular, the median of the group size distribution is 21. With 16 shelters, the first quartile of group size is 14 and the third quartile is 23, while with 25 shelters the inter-quartile extrema are 15 and 25.

The average behavior of the system over 80 runs, showed by the bottom plots of Figure 7.8, confirms these observations. The median of the formed groups grows to 22, and the extrema of the inter-quartile range of the final sizes I observed are 15 and 25.

The plots also show that the time to reach convergence is very similar with 16 and 25 shelters. The convergence time of the system appears to be practically independent of the number of robots, thus confirming that the proposed group formation mechanism displays the spreading property. This result is expected because the system is completely parallel and based on local interactions among mobile robots and shelters.

7.6.3 Spreading

To test spreading, I devised a two-phase experimental setting. In the first phase, the system is given 750 s to reach convergence. At the beginning of the second phase, I activate or deactivate a shelter. These events force the system to redistribute the mobile robots and reach a new convergence state, thus providing a good test of how capable the system is of spreading robots between shelters. The length of the second phase is set to 750 s.

As I explained in Section 7.5, upon activation and deactivation, a shelter broadcasts a signal to its neighbors. The neighbors of a shelter are those shelters in direct line of sight. The signal forces the recipients to reset the leaving probability to l^{hi} and to restart the decay process.

An important aspect in the system is the transmission range of the reset signal. In Section 7.6.3, I analyze the results I obtained in experiments in which the reset signal is broadcast globally throughout the system. In Section 7.6.3, I discuss the results I obtained when the transmission range of the signal is limited.

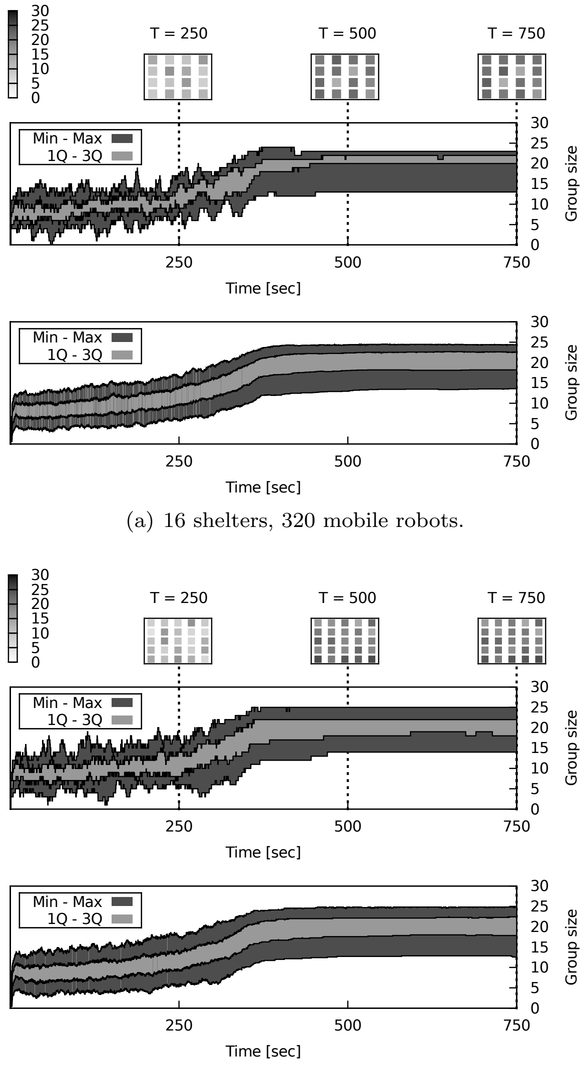
Global Reset Signal

In this section, I analyze the results I obtained with a global broadcast of the reset signal across the shelters. In these experiments I use 9 shelters and 180 robots.

To study whether the starting point of the (de)activation event affects the system behavior, I explore the cases in which the event occurs at the corner and at the center of the shelter grid.

The results are reported in Figures 7.9 and 7.10. In all the experimental settings, at the end of phase 1 the system reaches convergence to a state in which the group size distribution is tightly packed around the target value 20. The event causes the shelters to release the grouped robots. The subsequent dynamics, in the average plots, shows that, regardless of the location of the event and its type (activation/deactivation), the system is able to reach a new convergence state. The final distribution of the group sizes is tightly packed around 20 robots.

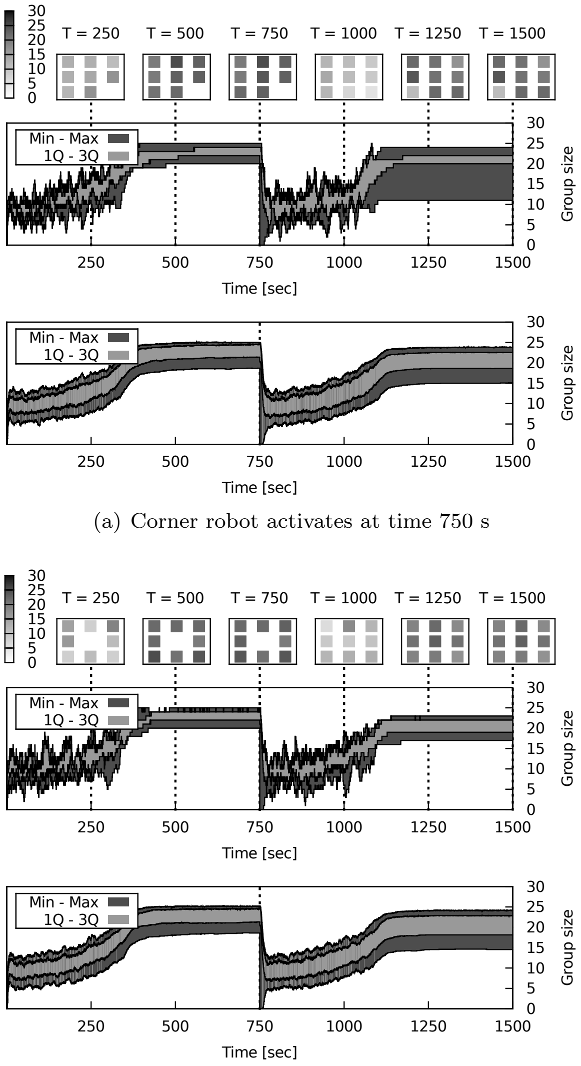
The snapshots in Figures 7.9 and 7.10 of the sample experiments show that the initial location of the event does not affect the final distribution of the robots. The gray levels in the snapshot do not display any visible bias towards a specific region of the group formation area, thus indicating that spreading is fair.



(a) 16 shelters, 320 mobile robots.

(b) 25 shelters, 500 mobile robots.

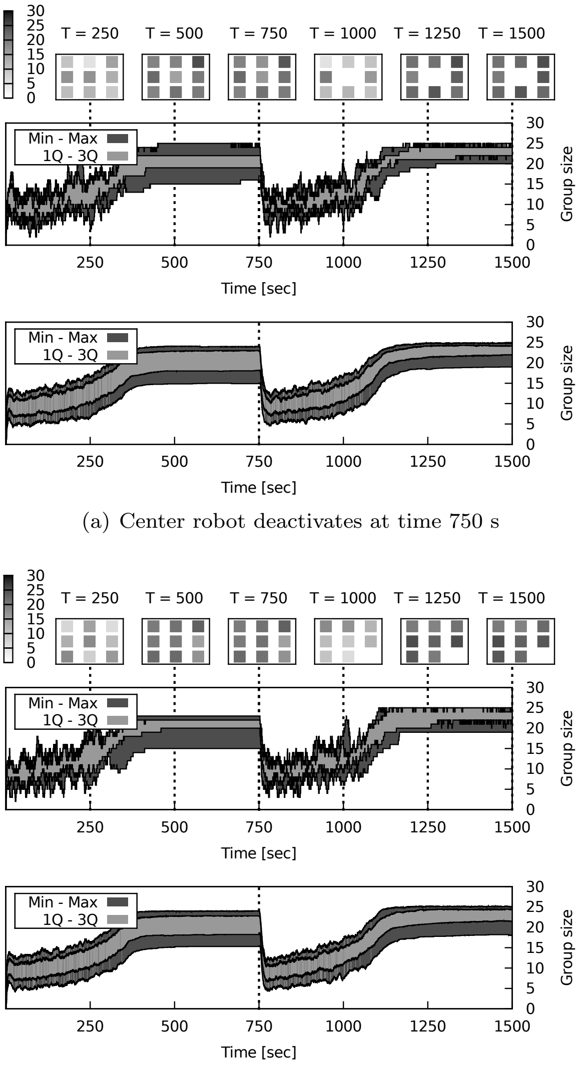
Figure 7.8: Scalability experiments testing the *convergence* and *spreading* properties of the system. Results are shown for two sets of experiments with 16 shelters (a) and 25 shelters (b). 80 experimental runs per set of experiments. The top plots show the behavior of the system in a single sample experiment that I have selected. The grids of squares represent snapshots of the state of the system at given moments in time during this sample experiment. The gray intensity of each individual square corresponds to the number of mobile robots recruited at that time by a single shelter. The min-max lines show the size of the largest recruited group of mobile robots and the size of the smallest recruited group of foot-bots at any given moment. The 1Q and 3Q lines show the inter-quartile range of the distribution of recruited group sizes among the shelters. The 1Q is the first quartile and shows the minimum recruited group size once I discard the lowest 25% of groups. The 3Q line is the third quartile, and shows the maximum recruited group size once I discard the highest 25% of the data. The bottom plot shows the same data averaged over all 80 runs.



(a) Corner robot activates at time 750 s

(b) Center robot activates at time 750 s

Figure 7.9: Set of experiments testing the *spreading* property of the system. All experiments run with 9 shelters and 180 mobile robots in a recruitment area consisting of a 3x3 shelter formation. Results are shown for two sets of experiments. Each experimental run lasts for 1,500 s. 20 experimental runs were conducted for each set of experiments. Top plots in each set represent selected sample runs, while bottom plots represent data averaged over all 20 runs. For a more detailed explanation of the plots see previous caption from Figure 7.8.



(a) Center robot deactivates at time 750 s

(b) Corner robot deactivates at time 750 s

Figure 7.10: Set of experiments testing the *spreading* property of the system. All experiments run with 9 shelters and 180 mobile robots in a recruitment area consisting of a 3x3 shelter formation. Results are shown for two sets of experiments. Each experimental run lasts for 1,500 s. 20 experimental runs were conducted for each set of experiments. Top plots in each set represent selected sample runs, while bottom plots represent data averaged over all 20 runs. For a more detailed explanation of the plots see previous caption from Figure 7.8.

Local Reset Signal

Although a global reset signal allows for fair spreading after the activation and deactivation of shelters, its application to real scenarios is problematic. In fact, if the frequency of these events is too high, the system may have insufficient time to reach convergence, resulting in constantly fluctuating group sizes. To prevent constant fluctuations, the convergence results of Section 7.6.2 suggest that the time between activation/deactivation events must be greater than the decay period δ. As the size of the scenario grows, and with it the number of group formation requests per time unit, the frequency of the events is likely to exceed δ^{-1}, thus causing constant fluctuations. A possible solution to this problem is to limit the range of transmission of the reset signal.

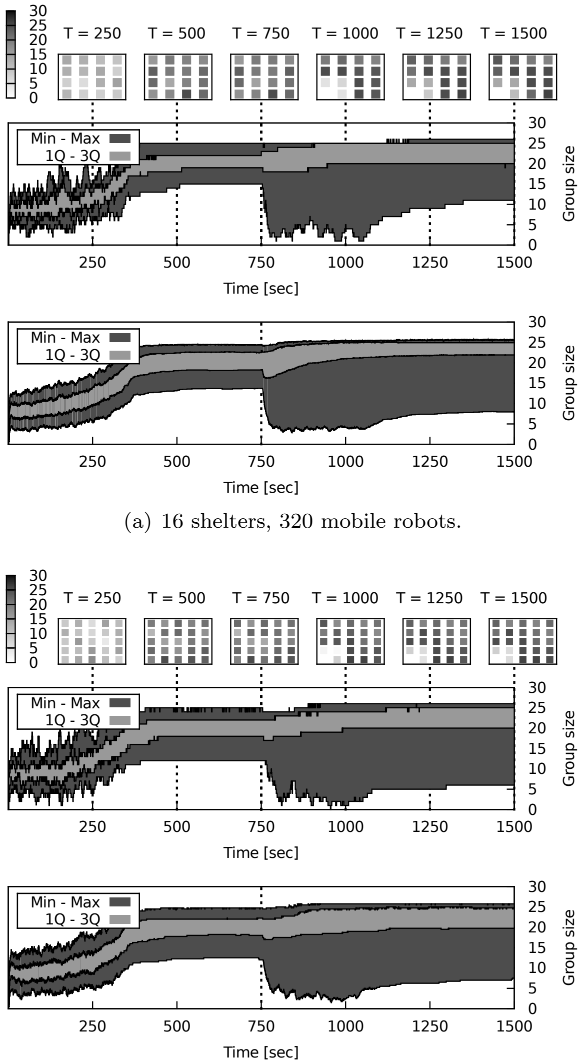
In this series of experiments, I test the impact on the spreading ability of the system when the reset signal is not globally broadcast. I use 16 and 25 shelters (to which correspond 320 and 500 mobile robots, respectively) and deactivate only one corner shelter.

Short range. Figure 7.11 shows the results of experiments in which the reset signal was transmitted only to the closest neighbors of a perturbed shelter (i.e., the shelters in the Moore neighborhood). The final state reached by the system presents two regions, one affected by the perturbation and one not affected by it. In the region affected by the perturbation, the final group sizes are visibly lower than in the other region. Thus, spreading is not fair. This phenomenon can be explained by observing that, upon leaving a group, the direction chosen by a robot is random and uniformly distributed. Thus, part of the robots leaked from the perturbed region to the unperturbed one.

Medium range. Figure 7.12 shows the results of experiments in which the reset signal was transmitted in a medium range. The reset signal reaches the neighbors of the neighbors of the originally perturbed shelter. Analogously to the previous case, the final state of the system is characterized by two regions, one affected by perturbation and one not affected. Since a larger part of the system takes part into the redistribution process, the distribution of the final group size is more even than the non-propagated signal case, but still spreading is not very fair.

Discussion

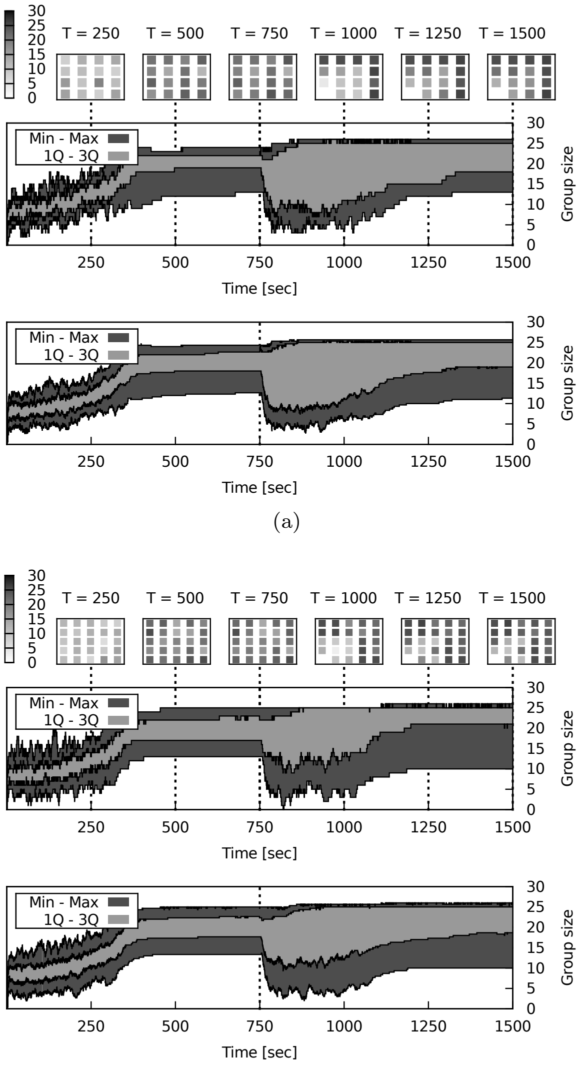
The communication range of the reset signal characterizes the ability of the system to redistribute the robots. The experiments show that, when the frequency of the perturbations exceeds δ^{-1}, limiting the range of the reset signal is not enough, by itself, to ensure fair redistribution. This problem is due to the fact that the robots that leave the perturbed



(a) 16 shelters, 320 mobile robots.

(b) 25 shelters, 500 mobile robots.

Figure 7.11: Set of experiments on local perturbation. In these experiments, the propagation of the reset signal is limited to the direct neighbors. Results are shown for two sets of experiments (a,b). 20 experimental runs were conducted for each set of experiments. Each experimental run lasts 1,500 s. Top plots in each set represent selected sample runs, while bottom plots represent data averaged over all 20 runs. For a more detailed explanation of the plots see previous caption from Figure 7.8.



(a)

(b)

Figure 7.12: Set of experiments on local perturbation. In these experiments, the propagation of the reset signal is limited to the second-level neighbors (i.e., the direct neighbors of the direct neighbors of the signal originator). Results are shown for two sets of experiments (a,b). 20 experimental runs were conducted for each set of experiments. Each experimental run lasts 1,500 s. Top plots in each set represent selected sample runs, while bottom plots represent data averaged over all 20 runs. For a more detailed explanation of the plots see previous caption from Figure 7.8.

shelters tend to leave the group in all the directions. As a consequence, only a part of the leaving robots are directed towards other perturbed shelters. The other robots join already stable groups. In this way, two regions are formed in the system—one region with a surplus of robots, and a region with a deficit of robots. A possible solution for this problem, currently under study, is to prevent the robots from navigating towards the unperturbed regions of the system.

Summary

In this chapter I presented a novel, distributed approach to the formation of multiple groups of mobile robots. The main feature of the algorithm is the ability to fairly distribute robots across group of different target sizes even in presence of scarce resources. The method takes inspiration from the behavior of cockroach aggregation under shelters. In the system, shelters are simple, active devices able to monitor the number of aggregated robots, and calculate and locally broadcast probabilities for the robots to join and leave the aggregate.

I showed that the stability of the groups is mainly dependent on the leaving probability, and presented a system in which shelters vary this probability over time to control group size. I demonstrated that, to achieve group size convergence and fair robot spreading, it is sufficient to let the leaving probability decay gradually from a high value to a low value.

I assessed the performance of the system in a large, challenging scenario in which hundreds of robots must be aggregated into dozens of groups. The results suggest that, due to its parallelism, the convergence time of the system is basically independent of the size of the scenario. Regarding spreading, the system is able to distribute the robots evenly among the shelters. In presence of perturbations, such as activation/deactivation of a shelter, spreading is fair if the time between two of these events is longer than the decay period of the leaving probabilities.

Future work involves studying methods to modify the system to enhance spreading with a high frequency of perturbations. A possible way to improve the system is to keep the propagation of the perturbations local, and prevent the robots that are temporarily freed from leaking to the non-perturbed region.

Chapter 8

Conclusions and Future Work

The long-term vision for swarm robotics applications is extremely diverse. Construction, search-and-rescue, space exploration, and surgery are but a few of the many applications foreseen in the distant future. At present, however, developing effective swarm robotics systems is a difficult endeavor due to the lack of general methodologies. Successful attempts to design these systems concentrate on specific scenarios, with little opportunity for generalization.

One of the most crucial hurdles to develop a general methodology for the design of swarm robotics systems is the lack of dedicated development tools. Development tools, such as simulators, programming languages, and robot platforms, constitute the backbone around which methodologies are conceived and validated.

This thesis deals with ARGoS, the first physics-based robot simulator specifically designed for swarm robotics applications. The faceted nature of swarm systems entails three main requirements: *(i)* accuracy, to ensure satisfactory correspondence between the results predicted by the simulation and the results obtained through real-world experiments; *(ii)* flexibility, to support any kind of robot and experiment; and *(iii)* efficiency, to minimize the time necessary to complete a simulation.

In the design of ARGoS, I have considered accuracy not as an intrinsic property of the simulator, but as a measurable quantity whose threshold of acceptance is decided by the experimenter. Consequently, in ARGoS accuracy is *tunable*. Accuracy tuning consists in choosing the models involved in a simulation. Models are encapsulated into modules, that the user selects as part of the configuration of the experiment. To this end, the architecture of ARGoS offers unprecedented levels of modularity, allowing the user to override any aspect of the simulation: sensors, actuators, physics engine, communication media, and visualizations.

Flexibility is a direct consequence of ARGoS' modularity. A unique feature of ARGoS is the possibility to execute the simulation using multiple physics engines, dedicating each engine to a portion of the physical space. In addition, ARGoS is implemented as a library with a powerful API, which renders embedding it into larger applications easy and fast. For instance, ARGoS has been interfaced to genetic algorithms to perform evolutionary robotics experiments.

Efficiency in ARGoS is obtained through a wide set of techniques and design choices that span multiple levels of the architecture. Tunable accuracy allows the user to assign computational resources to the relevant aspects of a simulation, and to employ less accurate (and, thus, faster) models for unimportant aspects. Differently from any existing simulator, in ARGoS, spatial indexing is implemented as a policy. This allows the user to select the best spatial index for the experiment at hand. Additionally, multi-threading ensures a good use of modern, multi-core architectures. A unique feature of ARGoS in this respect is the possibility to choose the scheduling strategy for assigning tasks to threads.

Experimental evaluation demonstrates that ARGoS is capable of simulating thousands of robots in a fraction of real time. In particular, experiments on an average computer show that ARGoS can complete a navigation experiment involving 10,000 robots in 60% of real-time —an unmatched result in the literature so far.

ARGoS is open source software released under the terms of the MIT license. Over the years, it has been employed for a large number of research works in swarm robotics including navigation, pattern formation, self-assembly, foraging, task allocation, and construction. At the moment of writing, ARGoS is used in 15 laboratories worldwide, and it is the official robot simulator of the projects Swarmanoid, ASCENS, E-SWARM, H2SWARM, and Swarmix. A little community of contributors is providing support in the form of bug reports and third-party extensions. For instance, ARGoS has been integrated by Kudelski et al. (2013) with the well-known network simulator ns3,[1] producing the first software capable of both simulating the physics and the WiFi communication capabilities of a robot swarm accurately.

Future work on ARGoS will follow several directions. One direction is to further improve its performance, possibly reaching hundreds of thousands of robots in real time. One possible way to achieve this result is integrating ARGoS with data structures that exploit the graphical processing unit (GPU) onboard any modern personal computer. Additionally, the architecture of ARGoS could be redesigned to be multi-process as well as multi-threaded. Multi-process architectures are already employed by several simulators, such as Gazebo and Morse. However, multi-processing in these simulators is implemented as a centralized system in which a special node coordinates the operations of the other nodes. To improve performance, a possible alternative could be a

[1] https://www.nsnam.org/

peer-to-peer architecture, in which nodes exchange information directly, without the need for a centralized coordinator.

Another direction for future work is integrating ARGoS with other tools, such as programming languages and analysis tools. Regarding languages, users can now program robot control code in C++, ASEBA, and Lua, and integration with languages such as Proto and ROS is planned. Furthermore, ARGoS will soon be integrated with the advanced statistical analyzer MultiVeStA (Sebastio and Vandin, 2013).

Last but not least, effort will be put in the integration of more robots and more experimental modalities. For instance, realistic models for underwater experiments, and voxel-based physics engines to simulate shores and construction with deformable materials are currently under study. Additionally, improved support for modular robots, magnetism, and articulated robots is currently being developed.

Looking forward, my hope is that ARGoS will play an increasingly important role in the development of swarm robotics. In particular, by its very nature, ARGoS is suitable to tackle research questions that have received little attention so far. Among these questions, I believe that three are paramount: *(i)* how to realize effective human-swarm interaction, that allows an operator to influence the behavior of an autonomous swarm in tasks such as search-and-rescue, exploration, and construction; *(ii)* how to develop, debug, and maintain large-scale swarm behaviors; and *(iii)* how to establish general benchmarks to characterize and compare existing approaches, thus fostering the emergence of best practices and solid methodologies.

Appendix A

Other Scientific Contributions

Over the course of my doctorate, I have conducted a number of studies not directly related to the main topic of this thesis. These works touch several topics, and can be divided into two categories: *swarm robotics* and *Boolean network robotics*.

A.1 Swarm Robotics

A.1.1 Pattern Formation and Flocking

- C. Pinciroli, M. Birattari, E. Tuci, M. Dorigo, M. Del Rey Zapatero, T. Vinko, D. Izzo. **Self-Organizing and Scalable Shape Formation for a Swarm of Pico Satellites**. *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2008).* IEEE-CS Press, Washington, DC, USA, pages 57–61, 2008.
- C. Pinciroli, M. Birattari, E. Tuci, M. Dorigo, M. Del Rey Zapatero, T. Vinko, D. Izzo. **Lattice Formation in Space for a Swarm of Pico Satellites**. *The Sixth International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS-2008).* Springer LNCS 5217, Berlin, Germany, pages 347–354, 2008.

Pattern formation is an activity whereby mobile robots displace themselves autonomously so as to form a predefined shape. In these works, we have explored the problem of forming patterns with small satellites orbiting a planet.

- A. Stranieri, E. Ferrante, A. E. Turgut, V. Trianni, C. Pinciroli, M. Birattari, M. Dorigo. **Self-Organized Flocking with a Heterogeneous Mobile Robot Swarm**. *Advances in Artificial Life (ECAL 2011).* MIT press, Cambridge, MA, pages 789–796, 2011.
- E. Ferrante, A. E. Turgut, C. Huepe, A. Stranieri, C. Pinciroli, M. Dorigo. **Self-Organized Flocking with a Mobile Robot**

Swarm: a Novel Motion Control Method. *Adaptive Behavior*, 20(6):460–477, 2012.

While pattern formation aims to create a static shape, the purpose of *flocking* is to coordinate the motion of a group of mobile robots towards a target location. As a result of coordination, the robot swarm moves as a unique entity.

An important research question is whether explicit alignment negotiation among the robots is required to achieve flocking. To shed light on this question, I have cooperated to two works. In the first, the robot swarm is divided in two groups, aligning and non-aligning. Aligning robots exchange information to negotiate a direction; non-aligning robots lack this capability. Results show that indeed flocking is possible even in presence of non-aligning robots, but flocking performance increases with the fraction of aligning ones. In the second work, we moved our attention to an overlooked aspect of the flocking mechanism—motion control. Motion control is the component that translates the proximal and alignment components into actual movement. We proposed a novel motion control rule, which increases flocking performance sensibly with non-aligning robots.

- E. Ferrante, A. E. Turgut, A. Stranieri, C. Pinciroli, M. Birattari, M. Dorigo. **A Self-Adaptive Communication Strategy for Flocking in Stationary and Non-Stationary Environments**. *Natural Computing*, 2014. In press.

In another work, we explored the case in which a small group of robots in a swarm possesses information regarding the target location, and must guide the rest of the swarm. We proposed an algorithm that allows the robots to flock in the following conditions: *(i)* the target location is constant over time; *(ii)* the target location changes over time; *(iii)* two groups of robots have information about contrasting target locations.

The experiments on flocking have been conducted with ARGoS, both in simulation and with real robots.

A.1.2 Cooperative Navigation

- F. Ducatelle, G. Di Caro, C. Pinciroli, F. Mondada, L. M. Gambardella. **Communication Assisted Navigation in Robotic Swarms: Self-Organization and Cooperation**. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*. IEEE Computer Society Press, Los Alamitos, CA, pages 4981–4988, 2011.

In this set of works, we investigated distributed methods to route individual robots towards a destination. All these method revolve around the idea that robots form a network. Depending on the position of the robots with respect to the target locations, these methods dictate how information flows in the network, in order to guide a robot to its destination.

We investigated the idea of guiding wheeled robots using a swarm of flying robots. The wheeled robots are unaware of the structure of the environment and must navigate between two locations. To optimize the path followed by the wheeled robots, the flying robots adapt their position over time to the motion of the wheeled robots. To the best of our knowledge, this method is the first example of self-organized cooperation in a physically heterogeneous swarm.

- F. Ducatelle, G. Di Caro, C. Pinciroli, L. M. Gambardella. **Self-Organised Cooperation between Robotic Swarms**. *Swarm Intelligence*, 5(2):73–96, 2011.
- F. Ducatelle, G. Di Caro, A. Förster, M. Bonani, M. Dorigo, S. Magnenat, F. Mondada, R. O'Grady, C. Pinciroli, P. Rétornaz, V. Trianni, L. M. Gambardella. **Cooperative Navigation in Robotic Swarms**. *Swarm Intelligence*, available at http://link.springer.com/article/10.1007%2Fs11721-013-0089-4#page-1, 2013.

In the above method, the wheeled robots use information from the flying robots to navigate. The fact that the flying robots adapt their position to the motion of the wheeled robots can be interpreted as a way, for the wheeled robots, to achieve a form of stigmergic communication.

In subsequent works, we studied methods based on other forms of communication to produce behaviors that achieve cooperative navigation. In particular, we concentrated on the problem of routing a homogeneous swarm of wheeled robots between two locations, maintaining the assumption that the robots do not possess nor construct an explicit representation of the environment.

We proposed a method inspired by routing algorithms in mobile ad hoc networks. Each robot maintains a routing table containing the shortest path to each known target location. The target locations are first discovered by the robots that can perceive it directly. The information on a target location is broadcast periodically and propagated by all robots throughout the network. This algorithm allows a single robot to be routed to a specific destination. In addition, when the entire swarm is required to navigate between two points, this algorithm autonomously converges to a chain-like formation between the two locations. This swarm behavior occurs also in presence of obstacles.

- Á. Gutiérrez, A. Campo, F. C. Santos, C. Pinciroli, M. Dorigo. **Social Odometry in Populations of Autonomous Robots**. *The Sixth International Conference on Ant Colony Optimization and*

Swarm Intelligence (ANTS-2008). Springer LNCS 5217, Berlin, Germany, pages 371–378, 2008.

In a further paper, we show that a similar chaining behavior can be obtained if robots exchange distance information estimated from odometry data. Since the error of this estimate increases with the distance from the target, the robots communicate the level of confidence along with the information on the distance. This paper shows that this simple mechanism is sufficient to achieve a stable chain between two target locations.

All the above experiments have been conducted with ARGoS, both in simulation and with real robots.

A.1.3 Self-Assembly

- R. O'Grady, C. Pinciroli, R. Groß, A. L. Christensen, F. Mondada, M. Bonani, M. Dorigo. **Swarm-bots to the Rescue**. *European Conference on Artificial Life (ECAL 2009).* Springer LNCS 5777, Berlin, Germany, pages 165–172, 2009.

In self-assembly, groups of robots connect to each other to perform tasks that would not be feasible individually.

One of today's open problems in self-assembly is when to trigger the self-assembling process. In this paper, we propose an algorithm that allows robots to decide between performing a task individually and cooperating by forming physical connections to other robots. We test our algorithm in rescue scenario in which broken robots must be transported to safety by rescuer robots. Each broken robot requires a different number of rescuers to be transported. The algorithm assumes that rescuers are not aware a priori of the required number of robots to complete a transport.

- R. O'Grady, A. L. Christensen, C. Pinciroli, M. Dorigo. **Robots Autonomously Self-Assemble into Dedicated Morphologies to Solve Different Tasks**. *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010).* International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pages 1517–1518, 2010.

In this work, we investigated another important open problem in self-assembly: Which morphology must be selected for a given task? The result of our work is a distributed algorithm that can autonomously select between three possible morphologies capable of solving tasks that appear to the robots in an unknown order.

A.1.4 Design of Artificial Swarm Systems

- M. Brambilla, C. Pinciroli, M. Birattari, M. Dorigo. **Property-driven Design for Swarm Robotics**. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012).* International Foundation for Autonomous Agents and Multiagent Systems, pages 139–146, 2012.

In this paper, we present a novel approach to the design of swarm robotics systems that tackles the problem of linking macroscopic properties to microscopic behaviors. The typical design process consists in a code-and-fix loop that revolves around the concept that the design must specify *how* a swarm behavior is achieved. In contrast, the idea behind our approach is to start from a specification of *what* the system must achieve—its *properties.* The resulting design process is factorized in four phases. The result of each subsequent phase is an increasingly detailed model of the system. The final result is a complete, executable specification of the system. At each phase, the model is expressed through a formalism that allows the designer to verify the dynamics of the model against the target properties of the system. The experiments in this paper have been conducted with ARGoS, both in simulation and with real robots.

- E. Gjondrekaj, M. Loreti, R. Pugliese, F. Tiezzi, C. Pinciroli, M. Brambilla, M. Birattari, M. Dorigo. **Towards a Formal Verification Methodology for Collective Robotic Systems**. *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM 2012).* Springer, Berlin, Germany, 7635:54–70, 2012.

To apply the property-driven design approach, it is required to identify suitable formalisms to express the properties of the system that enable verification. In this paper, we propose an approach to formal verification that captures the main features of a distributed foraging behavior. The experiments in this paper have been conducted with ARGoS in simulation.

- M. Puviani, C. Pinciroli, G. Cabri, L. Leonardi, F. Zambonelli. **Is Self-Expression Useful? Evaluation by a Case Study**. *IEEE 22nd International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2013)*, IEEE Press, Piscataway, NJ, pages 62–67, 2013.

Designing distributed systems that are able to adapt is one of the grand challenges of our field. A current trend in the research in distributed systems is the identification of adaptation patterns, i.e., recurrent behavioral schema that display a form of adaptivity. Self-expression is the ability of a system to *(i)* detect the need for a change in adaptation pattern, and *(ii)* switch to a different adaptation pattern that suits the new operational conditions. In this paper, we explore the impact of self-expression on the performance of a swarm of robots that performs a foraging task. The experiments in this paper have been conducted with ARGoS in simulation.

A.1.5 Task Allocation and Task Partitioning

One of the basic research topics in swarm robotics involves the relationship between the tasks to perform and the available robots. The study of this topic can be conducted along two complementary paths.

- A. Brutschy, G. Pini, C. Pinciroli, M. Birattari, M. Dorigo. **Self-organized Task Allocation to Sequentially Interdependent Tasks in Swarm Robotics**. *Autonomous Agents and Multi-Agent Systems*, 28(1):101–125, 2014.

The first path concerns the mechanisms to assign robots to a set of given tasks. In the vast majority of the papers on this subject, the tasks are assumed independent and always active. In this paper, we explore a scenario that has so far been overlooked in the literature: assigning robots to non-independent tasks. More specifically, in our work the tasks are sequentially interdependent. In practice, the robots must transport objects from a source location to a target destination. Every object, in order to reach its destination, must first be deposited in an area located in the middle of the arena. The purpose of this area is to parallelize the transportation task, potentially improving the throughput. During the experiment, a robot must choose whether to transport objects between the source and the middle of the arena, of between the middle and the destination. The proposed method does not require explicit communication among the robots; rather, it is based on the individual perception of delay experienced every time a robot waits for an object to be deposited in the middle area. Results show that our algorithm is able to reach a state in which throughput is near-optimal.

- G. Pini, A. Brutschy, C. Pinciroli, M. Dorigo, M. Birattari. **Autonomous Task Partitioning in Robot Foraging: an Approach Based on Cost Estimation**. *Adaptive Behavior*, 21(2):118–136, 2013.

The second research path deals with the dual problem, i.e., how to partition a task into parts that can be executed by the robots in parallel. In this paper, we propose a method to achieve efficient task partitioning in which the robots estimate the cost of performing a task. The experimental scenario consists in transporting objects from a source location to a target destination. The robots do not possess nor construct a map of the environment, and rely on odometry information to estimate their position with respect to the two locations. Thus, in this scenario, the cost of performing the transportation corresponds to the time spent navigating between the two locations. With the increase of the distance between source and target, the information based on odometry becomes noisier, the likelihood that a robot gets lost augments, and the cost associated with the task grows. As a consequence, the robots prefer to perform the task partially, leaving its completion to other robots. Results show that this method has beneficial effects on the throughput of objects reaching the destination.

All the above experiments have been conducted with ARGoS, both in simulation and with real robots.

A.1.6 Swarm Size Estimation

- M. Brambilla, C. Pinciroli, M. Birattari, M. Dorigo. **A Reliable Distributed Algorithm for Group Size Estimation with Minimal Communication Requirements**. *14th International Conference on Advanced Robotics (ICAR 2009).* Proceedings on CD-ROM, paper ID 137, 6 pages, 2009.

In this paper, we propose an algorithm that allows robots in a swarm to estimate the size of the swarm in a distributed fashion. The algorithm is loosely inspired by the signaling behavior of fireflies and crickets. The experiments in this paper have been conducted with ARGoS in simulation.

A.1.7 Collective Decision Making

Collective decision-making is a process whereby the members of a group decide on a course of action by consensus.

- M. A. Montes de Oca, E. Ferrante, A. Scheidler, C. Pinciroli, M. Birattari, M. Dorigo. **Majority-Rule Opinion Dynamics with Differential Latency: A Mechanism for Self-Organized Collective Decision-Making**. *Swarm Intelligence*, 5(3-4):305–327, 2011.

In this paper, we introduce an algorithm that allows a swarm of robots to make a collective choice between two actions that have the same effect, but different execution time. The algorithm assumes that the robots do not possess a priori information on the execution time of either action. The experimental setup involves an asymmetric loop-like arena in which a location is marked as source, and another is marked as destination. The task consists in transporting an object from source to destination. The object is too heavy for a single robot to transport, so teams of three robots are employed. The robots must negotiate the direction to follow to complete the transportation task. Both directions eventually lead to the destination, but one direction is much shorter than the other. Results show that the algorithm converges to the optimal choice for all the teams involved. The experiments in this paper have been conducted with ARGoS, both in simulation and with real robots.

- A. Campo, Á. Gutiérrez, S. Nouyan, C. Pinciroli, V. Longchamps, S. Garnier, M. Dorigo. **Artificial Pheromone for Path Selection by a Foraging Swarm of Robots**. *Biological Cybernetics*, 103(5):339–352, 2010.

In this paper, we studied the problem of path selection from insights inspired by ant pheromone trails. The robots communicate in a network and exchange messages that behave as virtual ants. The objective of the work is to route the messages so as to create paths between various locations in the environment marked by nearby robots. The proposed method converges to the creation of optimal paths. Experimentation is conducted with mathematical models and real robots.

A.1.8 Heterogeneous Swarm Robotics

- M. Dorigo, D. Floreano, L.M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy, D. Burnier, A. Campo, A. L. Christensen, A. Decugnière, G. Di Caro, F. Ducatelle, E. Ferrante, A. Förster, J. Guzzi, V. Longchamp, S. Magnenat, J. Martinez Gonzalez, N. Mathews, M. Montes de Oca, R. O'Grady, C. Pinciroli, G. Pini, P. Rétornaz, J. Roberts, V. Sperati, T. Stirling, A. Stranieri, T. Stützle, V. Trianni, E. Tuci, A.E. Turgut, F. Vaussard. **Swarmanoid: a Novel Concept for the Study of Heterogeneous Robotic Swarms**. *IEEE Robotics & Automation Magazine*, 20(4):60–71, 2013.

In this paper, we present the main findings in the FET project Swarmanoid (2006-2010). This project dealt with the design and implementation of the first heterogeneous swarm of robots capable of operating in 3D. Among the main results of the project, we list three novel robot platforms—the foot-bot, a ground-based robot; the eye-bot, a fully autonomous flying robot; and the hand-bot, a robot capable of climbing a bookshelf—, ARGoS, and a number of novel coordination algorithms.

A.1.9 Collective Transport

- A. Decugnière, B. Poulain, A. Campo, C. Pinciroli, B. Tartini, M. Osée, M. Dorigo, M. Birattari. **Enhancing the Cooperative Transport of Multiple Robots**. *The Sixth International Conference on Ant Colony Optimization and Swarm Intelligence*

(ANTS-2008). Springer LNCS 5217, Berlin, Germany, pages 307–314, 2008.

Object transport is often used as a testbed to assess the performance of many swarm algorithms. Often, the experimental scenarios involve an object whose weight exceeds the transport capabilities of a single robot. Thus, transport is performed by a team of cooperating robots. In this paper, we introduce a concept design for a robotic cart capable of collecting and storing multiple objects. The cart is conceived to be transported by a team of robots. Using the cart, mobile robots can transport a higher number of objects, thus improving the performance of the system.

A.2 Boolean Network Robotics

Boolean networks were introduced by Stuart Kauffmann as a simple model of gene regulatory networks. The study of this kind of networks revealed that, despite the simplicity of their implementation, Boolean networks exhibit complex dynamics. Depending on their configuration, the dynamics of these networks display very diverse features. Three regimes of behavior have been identified: *(i)* stationary, which occurs when the network dynamics converges to a small number of stable attractors, each robust to external perturbations; *(ii)* chaotic, in which the dynamics present a large number of attractors, and perturbations cause dramatic changes in the network behavior; and *(iii)* critical, in which the dynamics are formed by a small number of diverse attractors that allow the network to be both robust to small perturbations, and adaptive to large ones.

The richness of the dynamics, joined with the simplicity of implementing them, convinced us that Boolean networks could be fruitfully employed as robot control systems, similarly to the evolutionary robotics approach.

- A. Roli, M. Manfroni, C. Pinciroli, M. Birattari. **On the Design of Boolean Network Robots**. *Proceedings of EVOApplications 2011.* Springer LNCS 6624, Berlin, Germany, pages 43–52, 2011.
- A. Roli, S. Benedettini, M. Birattari, C. Pinciroli, R. Serra, M. Villani. **Robustness, Evolvability and Complexity in Boolean Network Robots**. *European Conference on Complex Systems (ECCS 2011).* Poster.

In a series of preliminary studies, we investigated the feasibility of this approach in simple phototaxis experiments involving an e-puck robot. Results demonstrated that indeed Boolean networks are capable of acting as a control system that reacts to basic external stimuli.

- A. Roli, S. Benedettini, M. Birattari, C. Pinciroli, R. Serra, M. Villani. **A Preliminary Study on BN-Robots' Dynamics**. *Proceedings of the Italian Workshop on Artificial Life and Evolutionary Computation (WIVACE 2012).* Published on CD, ISBN 978-88-903581-2-8, 2012.
- A. Roli, S. Benedettini, M. Birattari, C. Pinciroli, R. Serra, M. Villani. **State Space Properties of Boolean Networks Trained for Sequence Tasks**. *European Conference on Complex Systems (ECCS 2012).* Extended abstract and poster, 2012.
- A. Roli, M. Villani, R. Serra, L. Garattoni, C. Pinciroli, M. Birattari. **Identification of Dynamical Structures in Artificial Brains: An Analysis of Boolean Network Controlled Robots**. *AI*IA 2013.* Springer, Berlin, Germany, LNAI 8249, pages 324–335, 2013.
- S. Benedettini, M. Villani, A. Roli, R. Serra, M. Manfroni, A. Gagliardi, C. Pinciroli, M. Birattari. **Dynamical Regimes and Learning Properties of Evolved Boolean Networks**. *Neurocomputing*, 99:111–123, 2013.

In three subsequent studies, we analyzed the structure of the state space in which successfully trained Boolean networks operate. Our findings show that *(i)* attractors encode basic behaviors, *(ii)* Boolean networks are capable of solving tasks in which memory is required, and *(iii)* Boolean networks display effective learning capabilities.

- L. Garattoni, A. Roli, M. Amaducci, C. Pinciroli, M. Birattari. **Boolean Network Robotics as an Intermediate Step in the Synthesis of Finite State Machines for Robot Control**. *ECAL 2013*, MIT Press, Cambridge, MA, pages 783–790, 2013.

In a further paper, we demonstrated that the structure of the state space of non-trivial Boolean networks can be represented as a finite state machine. As a consequence, we argue that Boolean networks could be employed as a low-cost intermediate step towards the synthesis of finite state machines for robot control.

A.3 Other Publications

- M. A. Montes de Oca, J. Peña, T. Stützle, C. Pinciroli, M. Dorigo. **Heterogeneous Particle Swarm Optimizers**. *IEEE Congress on Evolutionary Computation (CEC 2009).* IEEE Press, Piscataway, NJ, pages 698–705, 2009.

This paper deals with particle swarm optimization, a class of metaheuristic algorithms inspired by Boyd's model of bird flocking. In these algorithms, the position of each particle in the space represents a candidate solution to the optimization problem. The classical implementation of this class of algorithms assumes a homogeneous set of particles. In this paper, we show that using different kinds of particles diversifies the exploration of the solution space, resulting in a lower number of iterations necessary to find good solutions.

Appendix B

Compiling and Installing ARGoS

B.1 Licensing

ARGoS is released under the terms of the MIT license:

> The MIT License (MIT)
>
> Copyright (c) 2014 Carlo Pinciroli
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B.2 Downloading ARGoS

You can download a binary package of ARGoS from `http://iridia.ulb.ac.be/argos/download.php`. Alternatively, you can download the development sources through git:

```
$ git clone https://github.com/ilpincy/argos3.git argos3
```

B.3 Compiling ARGoS

B.3.1 Requirements

If you downloaded the sources of ARGoS and want to compile its code, you need:

- A UNIX system (Linux or MacOSX; Microsoft Windows is not supported)
- `g++` $\geq$ 4.3 (on Linux)
- `clang` $\geq$ 3.1 (on MacOSX)
- `cmake` $\geq$ 2.6

If you want to compile the simulator, you need:

- `gsl` $\geq$ 1.15
- `FreeImage` $\geq$ 3.15

The OpenGL-based graphical visualization is compiled only if the following libraries are found:

- `Qt` $\geq$ 4.6
- `freeglut` $\geq$ 2.6.0
- `libxi-dev` (on Ubuntu and other Debian-based systems)
- `libxmu-dev` (on Ubuntu and other Debian-based systems)

If you want to create the Lua wrapper you need:

- `lua` $==$ 5.1

If you want to create the documentation you need:

- To create the API:
  - `Doxygen` $\geq$ 1.7.3
  - `Graphviz/dot` $\geq$ 2.28
- To create the HTML version of the README:
  - `asciidoc` $\geq$ 8.6.2

Debian

On Debian, you can install all of the necessary requirements with the following command:

```
$ sudo apt-get install libgsl0-dev libfreeimage-dev libqt4-dev \
  freeglut3-dev libxi-dev libxmu-dev liblua5.1-dev lua5.1 \
  doxygen graphviz-dev asciidoc
```

OpenSuse

On openSUSE 12.3, you can install all of the necessary requirements with the following commands:

```
$ sudo zypper ar -n openSUSE-12.3-Graphics \
  http://download.opensuse.org/repositories/graphics/openSUSE_12.3/ \
  graphics

$ sudo zypper refresh

$ sudo zypper install git cmake gcc gcc-c++ gsl-devel freeimage-devel \
  doxygen graphviz asciidoc lua51-devel libqt4-devel freeglut-devel \
  rpmbuild
```

Mac OSX

On Mac, you can install all of the necessary requirements using HomeBrew.[1] On the command line, type the following command:

```
$ brew install pkg-config cmake gsl libpng freeimage lua qt \
  docbook asciidoc graphviz doxygen
```

B.3.2 Compiling the code

The compilation of ARGoS is configured through CMake.[2]

Fast compilation instructions

Compiling the ARGoS simulator:

```
$ cd argos3
$ mkdir build_simulator
$ cd build_simulator
$ cmake ../src
$ make
```

Compiling ARGoS for a robot:

```
$ cd argos3
$ mkdir build_myrobot
$ cd build_myrobot
$ cmake -DARGOS_BUILD_FOR=myrobot ../src
$ make
```

Compiling the documentation:

[1] http://http://brew.sh/

[2] http://www.cmake.org/

```
$ cd argos3
$ cd build_simulator # or 'cd build_myrobot'
$ make doc
```

ARGoS sources under Eclipse

To use Eclipse with the ARGoS sources, you must have the CDT[3] installed. Optionally, you can also install CMakeEd[4] to modify the CMakeLists.txt files comfortably within Eclipse.

To configure the ARGoS sources for Eclipse, it is better to avoid compiling the code in a separate build directory (for more details, see http://www.vtk.org/Wiki/Eclipse_CDT4_Generator#Out-Of-Source_Builds). Thus, execute CMake as follows:

```
$ cd argos3
$ cmake -G "Eclipse CDT4 - Unix Makefiles" src/
```

Now open Eclipse. Click on File → Import..., select Existing project into workspace, and click on Next. Set the base argos3 directory as the root directory in the dialog that appears. Click on Next and you're ready to go.

B.3.3 Advanced compilation configuration

The compilation of ARGoS can be configured through a set of CMake options:

Variable	**Type**	**Meaning [default value]**
CMAKE_BUILD_TYPE	*STRING*	Build type (Debug, Release, etc.) [empty]
CMAKE_INSTALL_PREFIX	*STRING*	Install prefix (/usr, /usr/local, etc.) [/usr/local]
ARGOS_BUILD_FOR	*STRING*	Target of compilation (simulator or robot name) [simulator]
ARGOS_BUILD_NATIVE	*BOOLEAN*	Whether to use platform-specific instructions [OFF]
ARGOS_THREADSAFE_LOG	*BOOLEAN*	Use or not the thread-safe version of LOG/LOGERR. [ON]
ARGOS_DYNAMIC_LOADING	*BOOLEAN*	Compile (and use) dynamic loading facilities [ON]
ARGOS_USE_DOUBLE	*BOOLEAN*	Use double (ON) or float (OFF) [ON]
ARGOS_DOCUMENTATION	*BOOLEAN*	Create documentation (API, manpage, etc.) [ON]

You can pass the wanted values from the command line. For instance, if you wanted to set explictly all the default values, you would write:

```
$ cd argos3/build_simulator
$ cmake -DCMAKE_BUILD_TYPE=Debug \
```

[3] http://www.eclipse.org/cdt/

[4] http://cmakeed.sourceforge.net/

```
          -DCMAKE_INSTALL_PREFIX=/usr/local \
          -DARGOS_BUILD_FOR=simulator \
          -DARGOS_BUILD_NATIVE=OFF \
          -DARGOS_THREADSAFE_LOG=ON \
          -DARGOS_DYNAMIC_LOADING=ON \
          -DARGOS_USE_DOUBLE=ON \
          -DARGOS_DOCUMENTATION=ON \
          ../src
```

IMPORTANT. When `ARGOS_BUILD_FOR` is set to `simulator`, `ARGOS_THREADSAFE_LOG` and `ARGOS_DYNAMIC_LOADING` must be `ON`.

TIP. For production environments, it is recommended to compile ARGoS with `CMAKE_BUILD_TYPE` set to `Release`. If you want to debug ARGoS, it is recommended to set `CMAKE_BUILD_TYPE` to `Debug`. The other standard settings (empty and `RelWithDebInfo`) are supported but should be avoided.

TIP. If you want to squeeze maximum performance from ARGoS, along with compiling with `CMAKE_BUILD_TYPE` set to `Release`, you can also set `ARGOS_BUILD_NATIVE` to `ON`. This setting instructs the compiler to use the compiler flags `-march=native` and `-mtune=native`. The code will run faster because you use the entire instruction set of your processor, but the generated binaries won't be portable to computers with different processors.

B.4 Using ARGoS from the source tree

IMPORTANT. You can't install ARGoS system-wide and run the source version at the same time. If you intend to run ARGoS from the sources, you must uninstall it from the system.

B.4.1 Running the ARGoS simulator

If you don't want to install ARGoS on your system, you can run it from the sources tree. In the directory `build_simulator/` you'll find a bash script called `setup_env.sh`. Executing this script, you configure the current environment to run ARGoS:

```
$ cd argos3
$ cd build_simulator
$ . setup_env.sh     # or 'source setup_env.sh'
$ cd core
$ ./argos3 -q all    # this shows all the plugins recognized by
                     # \argos
```

If you execute ARGoS with the graphical visualization, you'll notice that icons and textures are missing. This is normal, as ARGoS by default looks for them in the default install location. To fix this, you need to edit the file `$HOME/.config/Iridia-ULB/ARGoS.conf` as follows:

```
[MainWindow]
#
# other stuff
#
icon_dir=/path/to/argos3/src/plugins/simulator/visualizations/\
qt-opengl/icons/
texture_dir=/path/to/argos3/src/plugins/simulator/visualizations/\
qt-opengl/textures/
#
# more stuff
#
```

B.4.2 Debugging the ARGoS simulator

You can debug the ARGoS code using `gdb`. Since the code in scattered across multiple directories, you need a `.gdbinit` file. Luckily for you, this file is created automatically when you compile ARGoS. To use it, you just need to remember to run the ARGoS simulator from the `build_simulator/core/` directory:

```
$ cd argos3/build_simulator/core
$ gdb ./argos3
```

B.5 Installing ARGoS from the compiled binaries

To install ARGoS after having compiled the sources, it is enough to write:

```
$ cd argos3
$ cd build_simulator # or 'cd build_myrobot'
$ make doc           # documentation is required!
$ sudo make install
```

Alternatively, one can create a package. To build all the packages supported by your system, run these commands:

```
$ cd argos3
$ git tag -a X.Y.Z-release # give the package a unique version
                           # the format must be as shown
                           # X       = version major
                           # Y       = version minor
                           # Z       = version patch
```

```
                              # release = a textual label
$ cd build_simulator          # or 'cd build_myrobot'
$ cmake .                     # let CMake read the newly set tag
$ make doc                    # documentation is required!
$ make                        # compile the code
$ sudo make package           # make the package
```

This typically creates a self-extracting `.tar.gz` archive, a `.tar.bz2` archive, a `.zip` archive, and a platform-specific archive (`.deb`, `.rpm`, or a MacOSX package). You can determine which packages to create by setting the variables CPACK_BINARY_DEB, CPACK_BINARY_RPM, CPACK_BINARY_STGZ, CPACK_BINARY_TBZ2, CPACK_BINARY_TGZ, CPACK_BINARY_TZ.

IMPORTANT. The creation of source packages through the command `make package_source` is not supported. An easier option is to install ARGoS from a package distributed at http://iridia.ulb.ac.be/argos/download.php.

Appendix C

An Example of ARGoS in Use

C.1 The Robot Control Code

C.1.1 Header File

```
/*
 * AUTHOR: Carlo Pinciroli <cpinciro@ulb.ac.be>
 *
 * An example diffusion controller for the foot-bot.
 *
 * This controller makes the robots behave as gas particles. The
 * robots go straight until they get close enough to another
 * robot, in which case they turn, loosely simulating an elastic
 * collision. The net effect is that over time the robots diffuse
 * in the environment.
 *
 * The controller uses the proximity sensor to detect obstacles
 * and the wheels to move the robot around.
 *
 * This controller is meant to be used with the XML files:
 *    experiments/diffusion_1.argos
 *    experiments/diffusion_10.argos
 */

#ifndef FOOTBOT_DIFFUSION_H
#define FOOTBOT_DIFFUSION_H

/*
 * Include some necessary headers.
 */
/* Definition of the CCI_Controller class. */
#include <argos3/core/control_interface/ci_controller.h>
/* Definition of the differential steering actuator */
#include <argos3/plugins/robots/generic/control_interface/ci_differential_steering_actuator.h>
/* Definition of the foot-bot proximity sensor */
#include <argos3/plugins/robots/foot-bot/control_interface/ci_footbot_proximity_sensor.h>

/*
 * All the ARGoS stuff in the 'argos' namespace.
 * With this statement, you save typing argos:: every time.
 */
using namespace argos;

```

```
/*
 * A controller is simply an implementation of the CCI_Controller
 * class.
 */
class CFootBotDiffusion : public CCI_Controller {

public:

   /* Class constructor. */
   CFootBotDiffusion();

   /* Class destructor. */
   virtual ~CFootBotDiffusion() {}

   /*
    * This function initializes the controller.
    * The 't_node' variable points to the <parameters> section in
    * the XML file in the <controllers>
    * <footbot_diffusion_controller> section.
    */
   virtual void Init(TConfigurationNode& t_node);

   /*
    * This function is called once every time step.
    * The length of the time step is set in the XML file.
    */
   virtual void ControlStep();

   /*
    * This function resets the controller to its state right after
    * the Init().
    * It is called when you press the reset button in the GUI.
    * In this example controller there is no need for resetting
    * anything, so the function could have been omitted. It's here
    * just for completeness.
    */
   virtual void Reset() {}

   /*
    * Called to cleanup what done by Init() when the experiment
    * finishes.
    * In this example controller there is no need for clean
    * anything up, so the function could have been omitted. It's
    * here just for completeness.
    */
   virtual void Destroy() {}

private:

   /* Pointer to the differential steering actuator */
   CCI_DifferentialSteeringActuator* m_pcWheels;
   /* Pointer to the foot-bot proximity sensor */
   CCI_FootBotProximitySensor* m_pcProximity;

   /*
    * The following variables are used as parameters for the
    * algorithm. You can set their value in the <parameters>
    * section of the XML configuration file, under the
    * <controllers><footbot_diffusion_controller> section.
    */

   /* Maximum tolerance for the angle between
```

```
    * the robot heading direction and
    * the closest obstacle detected. */
   CDegrees m_cAlpha;
   /* Maximum tolerance for the proximity reading between
    * the robot and the closest obstacle.
    * The proximity reading is 0 when nothing is detected
    * and grows exponentially to 1 when the obstacle is
    * touching the robot.
    */
   Real m_fDelta;
   /* Wheel speed. */
   Real m_fWheelVelocity;
   /* Angle tolerance range to go straight.
    * It is set to [-alpha,alpha]. */
   CRange<CRadians> m_cGoStraightAngleRange;

};

#endif
```

C.1.2 Implementation File

```
/* Include the controller definition */
#include "footbot_diffusion.h"
/* Function definitions for XML parsing */
#include <argos3/core/utility/configuration/argos_configuration.h>
/* 2D vector definition */
#include <argos3/core/utility/math/vector2.h>

/****************************************/
/****************************************/

CFootBotDiffusion::CFootBotDiffusion() :
   m_pcWheels(NULL),
   m_pcProximity(NULL),
   m_cAlpha(10.0f),
   m_fDelta(0.5f),
   m_fWheelVelocity(2.5f),
   m_cGoStraightAngleRange(-ToRadians(m_cAlpha),
                           ToRadians(m_cAlpha)) {}

/****************************************/
/****************************************/

void CFootBotDiffusion::Init(TConfigurationNode& t_node) {
   /*
    * Get sensor/actuator handles
    *
    * The passed string (ex. "differential_steering") corresponds
    * to the XML tag of the device whose handle we want to have.
    * For a list of allowed values, type at the command prompt:
    *
    * $ argos3 -q actuators
    *
    * to have a list of all the possible actuators, or
    *
    * $ argos3 -q sensors
    *
    * to have a list of all the possible sensors.
    *
    * NOTE: ARGoS creates and initializes actuators and sensors
    * internally, on the basis of the lists provided the
    * configuration file at the <controllers><footbot_diffusion>
    * <actuators> and <controllers><footbot_diffusion><sensors>
    * sections. If you forgot to list a device in the XML and then
    * you request it here, an error occurs.
    */
   m_pcWheels    = GetActuator<CCI_DifferentialSteeringActuator>("differential_steering");
   m_pcProximity = GetSensor  <CCI_FootBotProximitySensor      >("footbot_proximity"    );
   /*
    * Parse the configuration file
    *
    * The user defines this part. Here, the algorithm accepts
    * three parameters and it's nice to put them in the config
    * file so we don't have to recompile if we want to try other
    * settings.
    */
   GetNodeAttributeOrDefault(t_node, "alpha", m_cAlpha, m_cAlpha);
   m_cGoStraightAngleRange.Set(-ToRadians(m_cAlpha), ToRadians(m_cAlpha));
   GetNodeAttributeOrDefault(t_node, "delta", m_fDelta, m_fDelta);
   GetNodeAttributeOrDefault(t_node, "velocity", m_fWheelVelocity, m_fWheelVelocity);
}

/****************************************/
/****************************************/

void CFootBotDiffusion::ControlStep() {
   /* Get readings from proximity sensor */
   const CCI_FootBotProximitySensor::TReadings& tProxReads = m_pcProximity->GetReadings();
   /* Sum them together */
   CVector2 cAccumulator;
   for(size_t i = 0; i < tProxReads.size(); ++i) {
      cAccumulator += CVector2(tProxReads[i].Value, tProxReads[i].Angle);
   }
   cAccumulator /= tProxReads.size();
   /* If the angle of the vector is small enough and the closest
    * obstacle is far enough, continue going straight, otherwise
    * curve a little.
    */
   CRadians cAngle = cAccumulator.Angle();
   if(m_cGoStraightAngleRange.WithinMinBoundIncludedMaxBoundIncluded(cAngle) &&
      cAccumulator.Length() < m_fDelta ) {
      /* Go straight */
      m_pcWheels->SetLinearVelocity(m_fWheelVelocity, m_fWheelVelocity);
   }
   else {
      /* Turn, depending on the sign of the angle */
      if(cAngle.GetValue() > 0.0f) {
         m_pcWheels->SetLinearVelocity(m_fWheelVelocity, 0.0f);
      }
      else {
         m_pcWheels->SetLinearVelocity(0.0f, m_fWheelVelocity);
      }
   }
```

```
}

/****************************************/
/****************************************/

/*
 * This statement notifies ARGoS of the existence of the
 * controller.
 * It binds the class passed as first argument to the string
 * passed as second argument.
 * The string is then usable in the configuration file to refer
 * to this controller.
 * When ARGoS reads that string in the configuration file, it
 * knows which controller class to instantiate.
 * See also the configuration files for an example of how this
 * is used.
 */
REGISTER_CONTROLLER(CFootBotDiffusion, "footbot_diffusion_controller")
```

C.2 The Experiment Configuration File

C.2.1 Single-Robot Experiment

```
<?xml version="1.0" ?>
<argos-configuration>

  <!-- ************************* -->
  <!-- * General configuration * -->
  <!-- ************************* -->
  <framework>
    <!--
        System configuration:
        - threads: the number of slave threads to parallelize the
          computation. For less than 100 robots thread management
          is not beneficial, so here we set it to 0. When set to
          0, it means that the computation is not parallelized:
          the main thread does everything.
    -->
    <system threads="0" />
    <!--
        Experiment configuration:
        - length: total experiment time in seconds (0 means
          the experiment has no time limit)
        - ticks_per_second: number of ticks per second
          (int value)
        - random_seed: seed of the main random number
          generator. If unset or set to zero, this value is
          taken from the clock and a warning message is
          displayed.
    -->
    <experiment length="0"
                ticks_per_second="10"
                random_seed="124" />
  </framework>

  <!-- *************** -->
  <!-- * Controllers * -->
  <!-- *************** -->
  <controllers>
```

```

    <!--
         Here you list the controllers to be used in the
         experiment.
         The XML tag is set by the

         REGISTER_CONTROLLER(class, "tag") macro.

         You find it in the .cpp file of your controller.
         For this example, the macro is called in
         controllers/footbot_diffusion.cpp:100.
    -->

    <!--
        The attributes are:
        - id: a unique a identifier for this controller, to be
          used in the subsequent <arena> section to say which
          robots use which controller
        - library: the path to the compiled library containing
          your controller.
    -->
    <footbot_diffusion_controller id="fdc"
                                  library="build/controllers/footbot_diffusion/libfootbot_diffusion.so">
      <!--
          The <actuators> section contains a list of the actuators
          used by this controller.
          If you forget a to mention an actuator here and then
          request it in the controller, an error occurs.
          For a list of the possible actuators, type at the
          command prompt:

          $ launch_argos -q actuators

          Multiple implementations of an actuator are possible. To
          identify which one you want to use, pass it in the
          'implementation' attribute below. When you type the
          'argos3 -q' command, the implementation is in the square
          brackets following the name of the device:

          $ argos3 -q actuators
          ...
          footbot_wheels [default]
          ...
      -->
      <actuators>
        <differential_steering implementation="default" />
      </actuators>
      <!--
          The <sensors> section contains a list of the sensors
          used by this controller.
          If you forget a to mention a sensor here and then
          request it in the controller, an error occurs.
          For a list of the possible sensors, type at the command
          prompt:

          $ argos3 -q sensors
      -->
      <sensors>
        <footbot_proximity implementation="default" show_rays="true" />
      </sensors>
      <!--
          The <params> section is passed as-is to the controller
          Init() function.
          The user, writing the controller, defines how it is
          organized.
          To understand what these parameters are for, check the
          controller header file in
          controllers/footbot_diffusion/footbot_diffusion.h.
      -->
      <params alpha="7.5" delta="0.1" velocity="5" />
    </footbot_diffusion_controller>

  </controllers>

  <!-- ********************** -->
  <!-- * Arena configuration * -->
  <!-- ********************** -->
  <!--
      Here you place all the objects in the arena.
      All linear measures are expressed in meters.
      Angles are expressed in degrees.
      The 'size' attribute contains the size of the arena around
      the origin.
      To get help about which entities are available, type at the
      command prompt:

      $ argos3 -q entities

      and to get help about a specific entity (for instance,
      the box)

      $ argos3 -q box
  -->
  <arena size="3, 3, 1" center="0,0,0.5">

    <!-- Place four boxes in a square to delimit the arena -->
    <box id="wall_north" size="2,0.1,0.5" movable="false">
      <body position="0,1,0" orientation="0,0,0" />
    </box>
    <box id="wall_south" size="2,0.1,0.5" movable="false">
      <body position="0,-1,0" orientation="0,0,0" />
    </box>
    <box id="wall_east" size="0.1,2,0.5" movable="false">
      <body position="1,0,0" orientation="0,0,0" />
    </box>
    <box id="wall_west" size="0.1,2,0.5" movable="false">
      <body position="-1,0,0" orientation="0,0,0" />
    </box>

    <!-- Place a foot-bot in the origin and bind it to the
         controller -->
    <foot-bot id="fb_0">
      <body position="0,0,0" orientation="0,0,0" />
      <controller config="fdc"/>
    </foot-bot>

  </arena>

  <!-- ****************** -->
  <!-- * Physics engines * -->
  <!-- ****************** -->
  <!--
      In ARGoS, multiple physics engines can run at the same time.
      In this section you say which engines to use for the
      experiment.
      To know which engines are available, type at the command
      prompt:

      $ argos3 -q physics_engines
  -->
  <physics_engines>
    <!--
        Use a 2D dynamics engine.
    -->
    <dynamics2d id="dyn2d" />
  </physics_engines>

  <!-- ********* -->
  <!-- * Media * -->
  <!-- ********* -->
  <!--
      Here you specify the media in use. Media allow robots to
      communicate.
      In this experiment, robots do not communicate, so no media
      are specified.
      To know which media are available, type at the command
      prompt:

      $ argos3 -q media
  -->
  <media />

  <!-- ****************** -->
  <!-- * Visualization * -->
  <!-- ****************** -->
  <!--
      Here you specify which visualization to use.
      You can also not specify a visualization at all, in which
      case ARGoS will run without showing anything.
      Having no visualization is useful when you run ARGoS in a
      batch of experiments to collect statistics.
      To know which visualizations are available, type at the
      command prompt:

      $ argos3 -q visualizations
  -->
  <visualization>
    <qt-opengl />
  </visualization>

</argos-configuration>
```

C.2.2 Multi-Robot Experiment

```
<?xml version="1.0" ?>
<argos-configuration>

  <!-- ************************* -->
  <!-- * General configuration * -->
  <!-- ************************* -->
  <framework>
  <system threads="0" />
  <experiment length="0"
              ticks_per_second="10"
              random_seed="124" />
</framework>

<!-- *************** -->
<!-- * Controllers * -->
<!-- *************** -->
<controllers>

  <footbot_diffusion_controller id="fdc"
                                library="build/controllers/
                                    footbot_diffusion/
                                    libfootbot_diffusion.so">
    <actuators>
      <differential_steering implementation="default" />
    </actuators>
    <sensors>
      <footbot_proximity implementation="default"
                         show_rays="true" />
    </sensors>
    <params alpha="7.5" delta="0.1" velocity="5" />
  </footbot_diffusion_controller>

</controllers>

<!-- *********************** -->
<!-- * Arena configuration * -->
<!-- *********************** -->
<arena size="5, 5, 1" center="0,0,0.5">

  <box id="wall_north" size="4,0.1,0.5" movable="false">
    <body position="0,2,0" orientation="0,0,0" />
  </box>
  <box id="wall_south" size="4,0.1,0.5" movable="false">
    <body position="0,-2,0" orientation="0,0,0" />
  </box>
  <box id="wall_east" size="0.1,4,0.5" movable="false">
    <body position="2,0,0" orientation="0,0,0" />
  </box>
  <box id="wall_west" size="0.1,4,0.5" movable="false">
    <body position="-2,0,0" orientation="0,0,0" />
  </box>

  <!--
      You can distribute entities randomly. Here, we distribute
      10 foot-bots in this way:
      - the position is uniformly distributed
        on the ground, in the square whose corners are
        (-2,-2) and (2,2)
      - the orientations are non-zero only when rotating around
        Z and chosen from a gaussian distribution, whose mean is
        zero degrees and standard deviation is 360 degrees.
  -->
  <distribute>
    <position method="uniform" min="-2,-2,0" max="2,2,0" />
    <orientation method="gaussian"
                 mean="0,0,0" std_dev="360,0,0" />
    <entity quantity="10" max_trials="100">
      <foot-bot id="fb">
        <controller config="fdc" />
        </foot-bot>
      </entity>
    </distribute>

    <!--
        We distribute 5 boxes uniformly in position and rotation
        around Z.
    -->
    <distribute>
      <position method="uniform" min="-2,-2,0" max="2,2,0" />
      <orientation method="uniform" min="0,0,0" max="360,0,0" />
      <entity quantity="5" max_trials="100">
        <box id="b" size="0.3,0.3,0.5" movable="false" />
      </entity>
    </distribute>

    <!--
        We distribute cylinders uniformly in position and with
        constant rotation (rotating a cylinder around Z does not
        matter)
    -->
    <distribute>
      <position method="uniform" min="-2,-2,0" max="2,2,0" />
      <orientation method="constant" values="0,0,0" />
      <entity quantity="5" max_trials="100">
        <cylinder id="c"
                  height="0.5"
                  radius="0.15"
                  movable="false" />
      </entity>
    </distribute>

  </arena>

  <!-- ****************** -->
  <!-- * Physics engines * -->
  <!-- ****************** -->
  <physics_engines>
    <dynamics2d id="dyn2d" />
  </physics_engines>

  <!-- ********* -->
  <!-- * Media * -->
  <!-- ********* -->
  <media />

  <!-- ****************** -->
  <!-- * Visualization * -->
  <!-- ****************** -->
  <visualization>
    <qt-opengl />
  </visualization>

</argos-configuration>
```

Bibliography

A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*, chapter Policy-Based Class Design. C++ In Depth Series. Addison-Wesley, 2001.

J.-F. Allan. Robotics for distribution power lines: Overview of the last decade. In *2nd International Conference on Applied Robotics for the Power Industry (CARPI)*, pages 96–101. IEEE Press, Piscataway, NJ, Sept. 2012. ISBN 978-1-4673-4587-3. doi: 10.1109/CARPI.2012.6473344.

J.-M. Amé, J. Halloy, C. Rivault, C. Detrain, and J.-L. Deneubourg. Collegial decision making based on social amplification leads to optimal group formation. *Proceedings of the National Academy of Sciences of the United States of America*, 103(15):5835–5840, Apr. 2006. ISSN 0027-8424. doi: 10.1073/pnas.0507877103. URL http://dx.doi.org/10.1073/pnas.0507877103.

E. Aro. The utility of an autonomous multi-robot system of underwater floats. In *2nd IFAC Workshop on Multivehicle Systems*, pages 55–59. IFAC, 2012.

M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2794–2800. IEEE Press, Piscataway, NJ, Oct. 2007. ISBN 978-1-4244-0911-2. doi: 10.1109/IROS.2007.4399480.

M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In P. M. Hill and D. S. Warren, editors, *Proceedings of the International Conference on Logic Programming (ICLP '09)*, pages 265–280. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-02846-5_24.

M. P. Ashley-Rollman, P. Pillai, and M. L. Goodstein. Simulating multi-million-robot ensembles. In *2011 IEEE International Conference on Robotics and Automation*, pages 1006–1013, Shanghai, China, May

2011. IEEE Press, Piscataway, NJ. ISBN 978-1-61284-386-5. doi: 10.1109/ICRA.2011.5979807.

J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35 (2):97–113, 2003. doi: 10.1145/857076.857077.

J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, May 2010. ISSN 0941-0643. doi: 10.1007/s00521-010-0382-8.

S. Balakirsky and E. Messina. MOAST and USARSim - A Combined Framework for the Development and Testing of Autonomous Systems. In *Proceedings of the 2006 SPIE Defense and Security Symposium*, 2006.

S. Balakirsky, F. Proctor, C. Scrapper, and T. Kramer. A Mobile Robot Control Framework: From Simulation to Reality. In S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pages 111–122, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89075-1. doi: 10.1007/978-3-540-89076-8.

S. Balakirsky, S. Carpin, G. Dimitoglou, and B. Balaguer. From Simulation to Real Robots with Predictable Results: Methods and Examples. In R. Madhavan, E. Tunstel, and E. Messina, editors, *Performance Evaluation and Benchmarking of Intelligent Systems*, chapter 6, pages 113–137. Springer New York Dordrecht Heidelberg London, 2009. ISBN 978-1-4419-0491-1. doi: 10.1007/978-1-4419-0492-8.

T. Balch, F. Dellaert, A. Feldman, A. Guillory, C. Isbell, Z. Khan, A. Stein, and H. Wilde. How Multirobot Systems Research will Accelerate our Understanding of Social Animal Behavior. *Proceedings of the IEEE*, 94(7):1445–1463, 2006. doi: 10.1109/JPROC.2006.876969.

J. Beal and J. Bachrach. Infrastructure for on Sensor / Actuator Networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006. doi: 10.1109/MIS.2006.29.

R. Beckers, O. E. Holland, and J.-L. Deneubourg. From Local Actions to Global Tasks: Stigmergy and Collective Robotics. In R. A. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 181—-189. MIT Press Cambridge, MA, 1994.

G. Beni. From Swarm Intelligence to Swarm Robotics. *Swarm Robotics*, 3342:1–9, 2005.

E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, New York, NY, 1999.

M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada. The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4187–4193. IEEE Press, Piscataway, NJ, 2010.

M. Brambilla, C. Pinciroli, M. Birattari, and M. Dorigo. A reliable distributed algorithm for group size estimation with minimal communication requirements. In *Fourteenth International Conference on Advanced Robotics (ICAR 2009)*, pages 1–6, 2009. Proceedings on CD-ROM, paper ID 137.

M. Brambilla, C. Pinciroli, M. Birattari, and M. Dorigo. Property-driven design for swarm robotics. In *AAMAS'12 - Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 139–146. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, 2012.

M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, Jan. 2013. ISSN 1935-3812. doi: 10.1007/s11721-012-0075-2.

N. Bredeche, J.-M. Montanier, B. Weel, and E. Haasdijk. Roborobo! a Fast Robot Simulator for Swarm and Collective Robotics. Technical Report Ppsn, arXiv.org, 2013.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986. doi: 10.1109/JRA.1986.1087032.

K. G. C. Hongo, H. Kawamata. Catastrophic impact of typhoon waves on coral communities in the ryukyu islands under global warming. *Journal of Geophysical Research*, 117:2005–2012, 2012.

S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau. *Self-Organization in Biological Systems.* Princeton University Press, 2003.

Y. U. Cao, A. S. Fukunaga, and A. B. Kahng. Cooperative Mobile Robotics : Antecedents and Directions. *Autonomous Robots*, 4:1–23, 1997.

G. Caprari, P. Balmer, R. Piguet, and R. Siegwart. The autonomous micro robot "Alice": a platform for scientific and commercial applications. In *MHA'98. Proceedings of the 1998 International Symposium on Micromechatronics and Human Science*, pages 231–235. IEEE Press, Piscataway, NJ, 1998. ISBN 0-7803-5130-4. doi: 10.1109/MHS.1998.745787.

J. Carlson, R. Murphy, and A. Nelson. Follow-up analysis of mobile robot failures. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '04*, volume 5, pages 4987–4994. IEEE Press, Piscataway, NJ, 2004.

J. Carpenter. *The Quiet Professional: An investigation of U.S. military Explosive Ordnance Disposal personnel interactions with everyday field robots*. PhD dissertation, University of Washington, 2013.

S. Carpin, M. Lewis, J. Wang, and S. Balakirsky. USARSim : a robot simulator for research and education. In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation*, pages 1400–1405. IEEE Press, Piscataway, NJ, 2007a. ISBN 1424406021. doi: 10.1109/ROBOT.2007.363180.

S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. Bridging the Gap Between Simulation and Reality in Urban Search and Rescue. In G. Lakemeyer, E. Sklar, D. G. Sorrenti, and T. Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Computer Science*, pages 1–12, Berlin, Heidelberg, 2007b. Springer Berlin Heidelberg. ISBN 978-3-540-74023-0. doi: 10.1007/978-3-540-74024-7.

A. Castano, A. Behar, and P. M. Will. The CONRO modules for reconfigurable robots. *IEEE/ASME Transactions on Mechatronics*, 7(4): 403–409, 2002.

A. L. Christensen, R. O'Grady, and M. Dorigo. Morphology control in multirobot system. *IEEE Robotics and Automation Magazine*, 14(4): 18–25, 2007.

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

T. Collins, N. O. Ranasinghe, and W.-M. Shen. ReMod3D: A High-Performance Simulator for Autonomous, Self-Reconfigurable Robots. Technical Report Section I, University of South California, 2013.

N. Correll and A. Martinoli. System Identification of Self-Organizing Robotic Swarms. In M. Gini and R. Voyles, editors, *Distributed Autonomous Robotic Systems 7*, chapter 4, pages 31–40. Springer Japan, 2006. ISBN 978-4-431-35881-7. doi: 10.1007/4-431-35881-1_4.

E. Şahin. Swarm Robotics: From Sources of Inspiration to Domains of Application. In E. Şahin and W. M. Spears, editors, *Proceedings of the 2004 international conference on Swarm Robotics (SAB 2004)*, volume 3342, pages 10–20. Springer Berlin Heidelberg, 2005.

S. Curtis, J. Mica, J. Nuth, G. Marr, M. Rilee, and M. Bhat. ANTS (autonomous nano-technology swarm): An artificial intelligence approach to asteroid belt resource exploration. In *International Astronautical Federation*, 2000.

P. D'Arrigo and S. Santandrea. The APIES mission to explore the asteroid belt. *Advances in Space Research*, 38(9):2060–2067, 2006.

J.-L. Deneubourg, S. Goss, N. R. Franks, A. B. Sendova-Franks, C. Detrain, and L. Chrétien. The Dynamics of Collective Sorting Robot-Like Ants and Ant-Like Robots. In S. Wilson and J. A. Meyer, editors, *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, pages 356–363. MIT Press Cambridge, MA, 1990.

C. Detweiler, M. Vona, K. Kotay, and D. Rus. Hierarchical control for self-assembling mobile trusses with passive and active links. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '06*, pages 1483–1490. IEEE Press, Piscatway, NJ, 2006.

R. Diankov. *Automated construction of robotic manipulation programs*. PhD thesis, Carnegie Mellon University, Aug. 2010.

M. B. Dias, R. Zlot, N. Kalra, and A. Stentz. Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, 94(7): 1257–1270, 2006.

A. M. Donoghue. Occupational health hazards in mining: an overview. *Occupational Medicine*, 54(5):283–289, 2004.

M. Dorigo and M. Birattari. Swarm intelligence. *Scholarpedia*, 2(9): 1462, Jan. 2007. ISSN 0002-7979.

M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant algorithms and stigmergy. *Future Generation Computer Systems*, 16(8):851–871, 2000.

M. Dorigo, D. Floreano, L. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy,

D. Burnier, A. Campo, A. Christensen, A. Decugnière, G. Di Caro, F. Ducatelle, E. Ferrante, A. Förster, J. Guzzi, V. Longchamp, S. Magnenat, J. Martinez Gonzales, N. Mathews, M. Montes de Oca, R. O'Grady, C. Pinciroli, G. Pini, P. Rétornaz, J. Roberts, V. Sperati, T. Stirling, A. Stranieri, T. Stützle, V. Trianni, E. Tuci, A. Turgut, and F. Vaussard. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine*, 20(4):60–71, 2013.

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

F. Ducatelle, G. Di Caro, and L. Gambardella. Cooperative self-organization in a heterogeneous swarm robotic system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. ACM, New York, NY, 2010. Proceedings on CD-ROM.

F. Ducatelle, G. Di Caro, C. Pinciroli, and L. M. Gambardella. Self-organised cooperation between robotic swarms. *Swarm Intelligence*, 5 (2):73–96, 2011.

F. Ducatelle, G. Di Caro, A. Förster, M. Bonani, M. Dorigo, S. Magnenat, F. Mondada, R. O'Grady, C. Pinciroli, P. Rétornaz, V. Trianni, and L. M. Gambardella. Cooperative navigation in robotic swarms. *Swarm Intelligence*, 2013. http://link.springer.com/article/10.1007%2Fs11721-013-0089-4#page-1.

G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular open robots simulation engine: MORSE. In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. Ieee, May 2011. ISBN 978-1-61284-386-5. doi: 10.1109/ICRA.2011.5980252.

L. Edelstein-Keshet. Mathematical models of swarming and social aggregation. In *Proceedings of the 2001 International Symposium on Nonlinear Theory and its Applications*, pages 1–7, 2001.

D. Elizondo, T. Gentile, H. Candia, and G. Bell. Overview of robotic applications for energized transmission line work Technologies, field projects and future developments. In *1st International Conference on Applied Robotics for the Power Industry (CARPI 2010)*, pages 1–7. IEEE Press, Piscataway, NJ, Oct. 2010. ISBN 978-1-4244-6633-7. doi: 10.1109/CARPI.2010.5624478.

J. Everist, K. Mogharei, S. Harshit, N. Ranasinghe, B. Khoshnevis, P. Will, and W.-m. Shen. A system for in-space assembly. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*, volume 3, pages 2356–2361. IEEE Press, Piscataway, NJ, 2004. doi: 10.1109/IROS.2004.1389761.

J. G. Fernandez and A. Khademhosseini. Micro-masonry: Construction of 3d structures by microscale self-assembly. *Advanced Materials*, 22 (23):2538–2541, 2010.

E. Ferrante, A. E. Turgut, N. Mathews, M. Birattari, and M. Dorigo. Flocking in stationary and non-stationary environments: A novel communication strategy for heading alignment. In R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph, editors, *Parallel Problem Solving from Nature – PPSN XI*, volume 6239 of *Lecture Notes in Computer Science*, pages 331–340, Berlin, Germany, 2010. Springer Verlag.

E. Ferrante, A. E. Turgut, A. Stranieri, C. Pinciroli, M. Birattari, and M. Dorigo. A self-adaptive communication strategy for flocking in stationary and non-stationary environments. *Natural Computing*, 2013. In press.

M. Friedmann, K. Petersen, and O. von Stryk. Simulation of Multi-Robot Teams with Flexible Level of Detail. In S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pages 29–40, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89075-1. doi: 10.1007/978-3-540-89076-8.

T. Fukuda. Cell structured robotic system CEBOT: Control, planning and communication methods. *Robotics and Autonomous Systems*, 7 (2–3):239–248, 1991.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

S. Garnier. From Ants to Robots and Back : How Robotics Can Contribute to the Study of Collective Animal Behavior. In Y. Meng and Y. Jin, editors, *Bio-Inspired Self-Organizing Robotic Systems*, Studies in Computational Intelligence, chapter 5, pages 105–120. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-20760-0.

S. Garnier, C. Jost, R. Jeanson, J. Gautrais, M. Asadpour, G. Caprari, and G. Theraulaz. Aggregation behaviour as a source of collective decision in a group of cockroach-like robots. *Advances in Artificial Life*, 3630:169–178, 2005.

S. Garnier, C. Jost, J. Gautrais, M. Asadpour, G. Caprari, R. Jeanson, A. Grimal, and G. Theraulaz. The embodiment of cockroach aggregation behavior in a group of micro-robots. *Artificial life*, 14(4):387–408, Jan. 2008. ISSN 1064-5462. doi: 10.1162/artl.2008.14.4.14400.

S. Garnier, M. Combe, C. Jost, and G. Theraulaz. Do ants need to estimate the geometrical properties of trail bifurcations to find an efficient route? A swarm robotics test bed. *PLoS computational biology*, 9(3):e1002903, Mar. 2013. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1002903.

V. Gazi and B. Fidan. Coordination and control of multi-agent dynamic systems: Models and approaches. In E. Şahin, W. M. Spears, and A. F. T. Winfield, editors, *Swarm Robotics*, Lecture Notes in Computer Science, pages 71–102. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-71541-2. doi: 10.1007/978-3-540-71541-2_6.

V. Gazi and K. M. Passino. Stability analysis of swarms. *IEEE Transactions on Automatic Control*, 48(4):692–697, 2003.

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

B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323. IEEE Press, Piscataway, NJ, 2003.

D. Goldsmith. *Voyage to the Milky Way: The Future of Space Exploration*. TV Books, NY, 1999. ISBN 978-1575000466.

S. C. Goldstein, T. C. Mowry, J. D. Campbell, M. P. Ashley-Rollman, M. De Rosa, S. Funiak, J. F. Hoburg, M. E. Karagozler, B. Kirby, P. Lee, P. Pillai, J. R. Reid, D. D. Stancil, and M. P. Weller. Beyond Audio and Video: Using Claytronics to Enable Pario. *AI Magazine*, 30(2):29–45, 2009. doi: 10.1609/aimag.v30i2.2241.

P. Grassé. La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interprétation des termites constructeurs. *Insects Sociaux*, 6:41–83, 1959.

J. Green. Underground mining robot: A csir project. In *2012 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 1–6. IEEE, Piscataway, NJ, 2012.

J. Green and S. Plumb. Mobile robot competition. underground mining: A challenging application in mobile robotics. In *IEEE Africon 2011*, pages 1–6. IEEE, Piscataway, NJ, 2011.

R. Groß and M. Dorigo. Self-assembly at the macroscopic scale. *Proceedings of the IEEE*, 96(9):1490–1508, 2008.

a. Gutierrez, A. Campo, M. Dorigo, J. Donate, F. Monasterio-Huelin, and L. Magdalena. Open E-puck Range & Bearing miniaturized board for local communication in swarm robotics. In *2009 IEEE International Conference on Robotics and Automation (ICRA 2009)*, pages 3111–3116. IEEE Press, Piscataway, NJ, May 2009. ISBN 978-1-4244-2788-8. doi: 10.1109/ROBOT.2009.5152456.

M. K. Habib. Humanitarian Demining: Reality and the Challenge of Technology - The State of the Arts. *International Journal of Advanced Robotics Systems*, 4(2):151–172, 2007.

H. Hamann and T. Schmickl. Modelling the swarm: Analysing biological and engineered swarm systems. *Mathematical and Computer Modelling of Dynamical Systems*, 18(1):1–12, Feb. 2012. ISSN 1387-3954. doi: 10.1080/13873954.2011.601426.

H. Hamann and H. Wörn. A Framework of Space-Time Continuous Models for Algorithm Design in Swarm Robotics. *Swarm Intelligence*, 2(2-4):209–239, 2008. doi: 10.1007/s11721-008-0015-3.

S. Hettiarachchi and W. Spears. Distributed adaptive swarm for obstacle avoidance. *International Journal of Intelligent Computing and Cybernetics*, 2(4):644–671, 2009.

M. G. Hinchey, R. Sterritt, and C. Rouff. Swarms and Swarm Intelligence. *Computer*, 40(4):111–113, 2007. doi: 10.1109/MC.2007.144.

N. Hoff, R. Wood, and R. Nagpal. Effect of sensor and actuator quality on robot swarm algorithm performance. *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4989–4994, Sept. 2011. doi: 10.1109/IROS.2011.6094529.

D. W. Holmes, J. R. Williams, and P. Tilke. An events based algorithm for distributing concurrent tasks on multi-core architectures. *Computer Physics Communications*, 181(2):341–354, Feb. 2010. ISSN 00104655. doi: 10.1016/j.cpc.2009.10.009.

A. Howard, M. Matarić, and G. Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 299–308. Springer, New York, 2002.

M. A. Hsieh, A. Halász, S. Berman, and V. Kumar. Biologically inspired redistribution of a swarm of robots among multiple sites. *Swarm Intelligence*, 2(2-4):121–141, Sept. 2008. ISSN 1935-3812. doi: 10.1007/s11721-008-0019-z.

S. Huh, U. Lee, H. Shim, J.-B. Park, and J.-H. Noh. Development of an unmanned coal mining robot and a tele-operation system. In *2011 11th International Conference on Control, Automation and Systems (ICCAS)*, pages 31–35. IEEE, Piscataway, NJ, 2011.

D. Izzo, L. Pettazzi, and M. Ayre. Mission Concept for Autonomous on Orbit Assembly of a Large Reflector in Space. In *56th International Astronautical Congress*, pages IAC–05–D1.4.03, 2005.

J. B. C. Jackson, M. X. Kirby, W. H. Berger, K. A. Bjorndal, L. W. Botsford, B. J. Bourque, R. H. Bradbury, R. Cooke, J. Erlandson, J. A. Estes, T. P. Hughes, S. Kidwell, C. B. Lange, H. S. Lenihan, J. M. Pandolfi, C. H. Peterson, R. S. Steneck, M. J. Tegner, and R. R. Warner. Historical overfishing and the recent collapse of coastal ecosystems. *Science*, 293:629–637, 2001.

R. Jeanson, C. Rivault, J.-L. Denebourg, S. Blanco, R. Fournier, C. Jost, and G. Theraulaz. Self-organized aggregation in cockroaches. *Animal Behavior*, 69:169–180, 2004.

R. Jeanson, C. Rivault, J.-L. Deneubourg, S. Blanco, R. Fournier, C. Jost, and G. Theraulaz. Self-organized aggregation in cockroaches. *Animal Behaviour*, 69(1):169–180, Jan. 2005. ISSN 00033472. doi: 10.1016/j.anbehav.2004.02.009.

R. Johansson and A. Saffiotti. Navigating by stigmergy: A realization on an rfid floor for minimalistic robots. In *IEEE International Conference on Robotics and Automation (ICRA'09)*, pages 245–252. IEEE Press, Piscataway, NJ, 2009. doi: 10.1109/ROBOT.2009.5152737.

M. Jørgensen, E. H. Østergaard, and H. H. Lund. Modular ATRON: modules for a self-reconfigurable robot. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '04)*, pages 2068–2073. IEEE, Piscatway, NJ, 2004.

G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. GameBots: A Flexible Test Bedfor Multiagent Team Research. *Communications of the ACM*, 45 (1):43–45, 2002.

O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

N. Knowlton and J. Jackson. Shifting baselines, local impacts, and global change on coral reefs. *PLoS One*, 6:e54, 2008.

N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154. Ieee, 2004. ISBN 0-7803-8463-6. doi: 10.1109/IROS.2004.1389727.

T. Krajník, V. Vonásek, D. Fišer, and J. Faigl. AR-drone as a platform for robotic research and education. In D. Obdržálek and A. Gottscheber, editors, *Research and Education in Robotics - EUROBOT 2011*, pages 172–186. Springer-Verlag GmbH Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21975-7_16.

J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, Dec. 2006. ISSN 0929-5593. doi: 10.1007/s10514-006-9013-8.

A. S. Krishnamoorthi. Cooperative visitor: A template technique for visitor creation. http://www.artima.com/cppsource/cooperative_visitor.html, July 11th 2007. Accessed: 2014-02-09.

M. Kudelski, L. M. Gambardella, and G. a. Di Caro. RoboNetSim: An integrated framework for multi-robot and network simulation. *Robotics and Autonomous Systems*, 61(5):483–496, Feb. 2013. ISSN 09218890. doi: 10.1016/j.robot.2013.01.003.

J. Lächele, A. Franchi, H. Bülthoff, and P. Robuffo Giordano. SwarmSimX: Real-Time Simulation Environment for Multi-robot Systems. In I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 7628, pages 375–387. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-34327-8_34.

J. Lächele, M. Riedel, P. Robuffo Giordano, and A. Franchi. SwarmSimX and TeleKyb : Two ROS-integrated Software Frameworks for Single- and Multi-Robot Applications. In *Proceedings of the 2013 IEEE Conference on Robotics and Automation (ICRA 2013)*. IEEE Press, Piscataway, NJ, 2013.

K. Lerman, A. Martinoli, and A. Galstyan. A Review of Probabilistic Macroscopic Models. In E. Şahin and W. M. Spears, editors, *SAB 2004 International Workshop*, pages 143–152. Springer Berlin Heidelberg, 2005. doi: 10.1007/978-3-540-30552-1_12.

R. Y. M. Li and S. W. Poon. *Construction Safety*. Risk Engineering. Springer, Berlin Heidelberg, 2013.

S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996. ISBN 0-201-83454-5.

W. Liu, A. F. T. Winfield, J. Sa, J. Chen, and L. Dou. Towards Energy Optimization: Emergent Task Allocation in a Swarm of Foraging Robots. *Adaptive Behavior*, 15(3):289–305, Sept. 2007. ISSN 1059-7123. doi: 10.1177/1059712307082088.

S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada. ASEBA: A modular architecture for event-based control of complex robots. *IEEE/ASME Transactions on Mechatronics*, PP(99): 1–9, 2010.

A. Marjovi, J. Nunes, L. Marques, and A. de Almeida. Multi-robot exploration and fire searching. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)*, pages 1929–1934. IEEE Press, Piscataway, NJ, 2009. doi: 10.1109/IROS.2009.5354598.

A. Martinoli, K. Easton, and W. Agassounon. Modeling Swarm Robotic Systems: a Case Study in Collaborative Distributed Manipulation. *The International Journal of Robotics Research*, 23(4):415–436, Apr. 2004. ISSN 02783649. doi: 10.1177/0278364904042197.

N. Mathews, A. Christensen, E. Ferrante, R. O'Grady, and M. Dorigo. Establishing spatially targeted communication in a heterogeneous robot swarm. In *Proceedings of 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pages 939–946. IFAAMAS, Toronto, Canada, 2010.

C. Mavroidis and A. Ferreira. Nanorobotics: Past, Present, and Future. In C. Mavroidis and A. Ferreira, editors, *Nanorobotics*, pages 3–27. Springer New York, New York, NY, 2013. ISBN 978-1-4614-2118-4. doi: 10.1007/978-1-4614-2119-1.

J. McLurkin, A. J. Lynch, S. Rixner, T. W. Barr, A. Chou, K. Foster, and S. Bilstein. A Low-Cost Multi-robot System for Research, Teaching, and Outreach. In A. Martinoli, F. Mondada, N. Correll, G. Mermoud, M. Egerstedt, M. A. Hsieh, L. E. Parker, and K. Stø y, editors, *Distributed Autonomous Robotic Systems: The 10th International Symposium*, pages 597–609. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-32723-0_43.

L. Meeden. Bridging the gap between robot simulations and reality with improved models of sensor noise. In J. Koza, editor, *Proceedings of the Third Annual Genetic Programming Conference*, pages 824–831. Morgan Kauffmann Publishers, San Francisco, CA, 1998.

C. Melhuish, O. Holland, and S. Hoddell. Convoying: Using chorusing to form travelling groups of minimal agents. *Robotics and Autonomous Systems*, 28:207–216, 1999.

D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *Proc. of the International Conference on Robotics and Automation (ICRA 11)*, Shanghai, China, May 2011.

O. Michel. Webots: Professional Mobile Robot Simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, Dec. 2004.

M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Upper Saddle River, NJ, 1967.

S. Mitri, S. Wischmann, D. Floreano, and L. Keller. Using robots to understand social behaviour. *Biological reviews of the Cambridge Philosophical Society*, 88(1):31–9, Feb. 2013. ISSN 1469-185X. doi: 10.1111/j.1469-185X.2012.00236.x.

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

F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, J.-C. Zufferey, D. Floreano, and A. Martinoli. The e-puck , a Robot Designed for Education in Engineering. In P. J. S. Gonçalves, P. J. D. Torres, and C. M. O. Alves, editors, *Proceedings of Robotica 2009 – 9th Conference on Autonomous Robot Systems and Competitions*, volume 1, pages 59–65. IPCB, Castelo Branco, Portugal, 2006.

M. A. Montes de Oca, E. Ferrante, N. Mathews, M. Birattari, and M. Dorigo. Opinion dynamics for decentralized decision-making in a robot swarm. In M. Dorigo et al., editors, *Proceedings of the Seventh International Conference on Swarm Intelligence (ANTS 2010)*, LNCS 6234, pages 251–262. Springer, Berlin, Germany, 2010.

L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):Paper ID 19, 2011. doi: 10.1145/1922649.1922656.

S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: self-reconfigurable modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 7(4):431–441, 2002.

E. Mytilinaios, M. Desnoyer, D. Marcus, and H. Lipson. Designed and evolved blueprints for physical self-replicating machines. In *Proceedings of the 9th International Conference on the Simulation and Synthesis of Living Systems (Artificial Life IX)*, pages 15–20. MIT Press, Cambridge, MA, 2004.

S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines.* A Bradford Book, MIT Press, Cambridge, MA, 2004. ISBN 978-0262640565.

S. Nouyan, A. Campo, and M. Dorigo. Path formation in a robot swarm: Self-organized strategies to find your way home. *Swarm Intelligence*, 2(1):1–23, 2008.

R. O'Grady, R. Groß, F. Mondada, M. Bonani, and M. Dorigo. Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In *Advances in Artificial Life*, volume 3630 of *Lecture Notes in Computer Science*, pages 272–281. Springer, Berlin Heideleberg, 2005.

R. O'Grady, C. Pinciroli, A. L. Christensen, and M. Dorigo. Supervised group size regulation in a heterogeneous robotic swarm. In *Proceedings of ROBOTICA 2009 - 9th International Conference on Autonomous Robot Systems and Competitions*, pages 113–119. IPCB, Castelo Branco, Portugal, 2009. ISBN 978-972-99143-7-9.

R. O'Grady, C. Pinciroli, R. Groß, A. L. Christensen, F. Mondada, M. Bonani, and M. Dorigo. Swarm-bots to the rescue. In G. Kampis, I. Karsai, and E. Szathmry, editors, *Advances in Artificial Life: 10th European Conference (ECAL 2009)*, volume 5777 of *Lecture Notes in Artificial Intelligence*, pages 165–172. Springer, Berlin, Germany, 2011. doi: http://dx.doi.org/10.1007/978-3-642-21283-3_21.

M. Patil, T. Abukhalil, and T. Sobh. Hardware Architecture Review of Swarm Robotics System: Self-Reconfigurability, Self-Reassembly, and Self-Replication. *ISRN Robotics*, 2013:1–11, 2013. ISSN 2090-8806. doi: 10.5402/2013/849606.

K. Petersen, R. Nagpal, and J. Werfel. TERMES: An autonomous robotic system for three-dimensional collective construction. In H. Durrant-Whyte, N. Roy, and P. Abbeel, editors, *Robotics: Science and Systems Conference VII*, pages Paper ID 35, Proceedings on CD–ROM. MIT Press, Cambridge, MA, 2011.

C. Pinciroli, M. Birattari, E. Tuci, M. Dorigo, M. D. R. Zapatero, T. Vinko, and D. Izzo. Lattice formation in space for a swarm of pico satellites. In *Proceedings of the Sixth International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS-2008)*, number 5217 in Lecture Notes in Computer Science, pages 347–354. Springer, Berlin, Germany, September 2008.

C. Pinciroli, R. O'Grady, A. Christensen, and M. Dorigo. Self-organised recruitment in a heterogeneous swarm. In *The 14th International Con-*

ference on Advanced Robotics (ICAR 2009), page 8, 2009. Proceedings on CD-ROM, paper ID 176.

G. Pini, A. Brutschy, M. Frison, A. Roli, M. Dorigo, and M. Birattari. Task partitioning in swarms of robots: An adaptive method for strategy selection. *Swarm Intelligence*, 5(3–4):283–304, 2011. doi: http://dx.doi.org/10.1007/s11721-011-0060-1.

G. Pini, A. Brutschy, C. Pinciroli, M. Dorigo, and M. Birattari. Autonomous task partitioning in robot foraging: an approach based on cost estimation. *Adaptive Behavior*, 21(2):118–136, 2013.

P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open multi-methods for c++. In *Proceedings of the ACM 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 123–134. ACM New York, NY, 2007.

J. Reif and J. Wang. Social potential fields: a distributed behavioral control for autonomous robots. *Robotics and Autonomous Systems*, 27(3):171–194, 1999.

B. Rinkevich. Conservation of coral reefs through active restoration measures: recent approaches and last decade progress. *Environmental Science and Technology*, 39:4333–4342, 2005.

J. Roberts, T. Stirling, J. Zufferey, and D. Floreano. Quadrotor using minimal sensing for autonomous indoor flight. In *European Micro Air Vehicle Conference and Flight Competition (EMAV)*, 2007. Proceedings on CD-ROM.

J. Roberts, T. Stirling, J.-C. Zufferey, and D. Floreano. 2.5d infrared range and bearing system for collective robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)*. IEEE Press, Piscataway, NJ, 2009.

E. Rohmer, S. P. N. Singh, and M. Freese. V-REP : a Versatile and Scalable Robot Simulation Framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1321–1326. IEEE Press, Piscataway, NJ, 2013. doi: 10.1109/IROS.2013.6696520.

M. Rubenstein, C. Ahler, and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. *2012 IEEE International Conference on Robotics and Automation*, pages 3293–3298, May 2012. doi: 10.1109/ICRA.2012.6224638.

R. F. Rubio. Mining: The challenge knocks on our door. *Mine Water and the Environment*, 31(1):69–73, 2012.

M. Rust, J. Owens, and D. Reierson. *Understanding and controlling the german cockroach.* Oxford University Press, UK, 1995.

H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Data Management Series. Morgan Kaufmann, San Francisco, CA, 2006. ISBN 978-0-12-369446-1.

T. Schmickl. How to Engineer Robotic Organisms and Swarms? In Y. Meng and Y. Jin, editors, *Bio-Inspired Self-Organizing Robotic Systems*, number 216342 in Studies in Computational Intelligence, pages 25–52. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-20760-0_2.

F. Schweitzer. *Brownian Agents and Active Particles. On the Emergence of Complex Behavior in the Natural and Social Sciences.* Springer-Verlag, Berlin, Germany, 2003.

S. Sebastio and A. Vandin. MultiVeStA: Statistical model checking for discrete event simulators. Technical report, IMT Lucca, Italy, 2013.

B. Sellner, F. Heger, L. Hiatt, R. Simmons, and S. Singh. Coordinated multi-agent teams and sliding autonomy for large-scale assembly. *Proceedings of the IEEE*, 94:1425–1444, 2006.

M. Sivasankar and R. Durairaj. Brief Review on Nano Robots in Bio Medical Applications. *Advances in Robotics and Automation*, 1(1): 1–4, 2012. doi: 10.4172/ara.1000101.

W. M. Spears, D. F. Spears, J. C. Hamann, and R. Heil. Distributed, Physics-Based Control of Swarms of Vehicles. *Autonomous Robots*, 17 (2/3):137–162, Sept. 2004. ISSN 0929-5593. doi: 10.1023/B:AURO.0000033970.96785.f2.

K. Støy. Using situated communication in distributed autonomous mobile robots. In *Proceedings of the 7th Scandinavian Conference on Artificial Intelligence*, pages 44–52. IOS Press, 2001.

A. Stroupe, A. Okon, M. Robinson, T. Huntsberger, H. Aghazarian, and E. Baumgartner. Sustainable cooperative robotic technologies for human and robotic outpost infrastructure construction and maintenance. *Autonomous Robots*, 20:113–123, 2006.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998. ISBN 978-0262193986.

A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, New Jersey, NJ, second edition, 2001.

Y. Terada and S. Murata. Automatic modular assembly system and its distributed control. *International Journal of Robotics Research*, 27 (3–4):445–462, 2008.

M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the Vision, Modeling, and Visualization Conference*, pages 47–54. Aka, Heidelberg, Germany, 2003.

A. E. Turgut, H. Çelikkanat, F. Gökçe, and E. Şahin. Self-organized flocking in mobile robot swarms. *Swarm Intelligence*, 2(2-4):97–120, Aug. 2008. ISSN 1935-3812. doi: 10.1007/s11721-008-0016-2.

C. Ünsal, c. H. Kili c and P. K. Khosla. Modular self-reconfigurable bipartite robotic system: implementation and motion planning. *Autonomous Robots*, 10(1):23–40, 2001.

R. Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4):189–208, Aug. 2008. ISSN 1935-3812. doi: 10.1007/s11721-008-0014-4.

J. Werfel, Y. Bar-Yam, D. Rus, and R. Nagpal. Distributed construction by mobile robots with enhanced building blocks. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA 2006)*, pages 2787–2794. IEEE, Piscataway, NJ, 2006. doi: 10.1109/ROBOT.2006.1642123.

A. F. T. Winfield, J. Sa, M.-C. Fernández-Gago, and C. Dixon. On Formal Specification of Emergent Behaviours in Swarm Robotic Systems. *International Journal of Advanced Robotics Systems*, 2(4):363–370, 2005.

M. Yim, D. G. Duff, and K. D. Roufas. PolyBot: a modular reconfigurable robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 514–520. IEEE Computer Society Press, Washington, DC, 2000.

M. Yim, Y. Zhang, J. Lamping, and E. Mao. Distributed control for 3d metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.

M. Yim, Y. Zhang, and D. Duff. Modular robots. *IEEE Spectrum*, 39 (2):30–34, 2002.

G. Zachmann. Minimal hierarchical collision detection. In *Proceedings of the ACM symposium on Virtual reality software and technology (VRST '02)*, pages 121–128. ACM New York, NY, 2002. doi: 10.1145/585740.585761.