Swarm Intell (2012) 6:271–295
DOI 10.1007/s11721-012-0072-5

ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems

Carlo Pinciroli · Vito Trianni · Rehan O'Grady · Giovanni Pini · Arne Brutschy · Manuele Brambilla · Nithin Mathews · Eliseo Ferrante · Gianni Di Caro · Frederick Ducatelle · Mauro Birattari · Luca Maria Gambardella · Marco Dorigo

Received: 10 September 2012 / Accepted: 25 October 2012 / Published online: 16 November 2012

Abstract We present a novel multi-robot simulator named ARGoS. ARGoS is designed to simulate complex experiments involving large swarms of robots of different types. ARGoS is the first multi-robot simulator that is at the same time both efficient (fast performance with many robots) and flexible (highly customizable for specific experiments). Novel design choices in ARGoS have enabled this breakthrough. First, in ARGoS, it is possible to partition the simulated space into multiple sub-spaces, managed by different physics engines

C. Pinciroli (✉) · R. O'Grady · G. Pini · A. Brutschy · M. Brambilla · N. Mathews · E. Ferrante · M. Birattari · M. Dorigo
IRIDIA, CoDE, Université Libre de Bruxelles, 50 Avenue F. Roosevelt, CP 194/6, 1050 Bruxelles, Belgium
e-mail: cpinciro@ulb.ac.be

R. O'Grady
e-mail: rogrady@ulb.ac.be

G. Pini
e-mail: gpini@ulb.ac.be

A. Brutschy
e-mail: abrutsch@ulb.ac.be

M. Brambilla
e-mail: mbrambil@ulb.ac.be

N. Mathews
e-mail: nmathews@ulb.ac.be

E. Ferrante
e-mail: eferrant@ulb.ac.be

M. Birattari
e-mail: mbiro@ulb.ac.be

M. Dorigo
e-mail: mdorigo@ulb.ac.be

V. Trianni
ISTC, Consiglio Nazionale delle Ricerche, via San Martino della Battaglia 44, 00185 Roma, Italy
e-mail: vito.trianni@istc.cnr.it

running in parallel. Second, ARGoS' architecture is multi-threaded, thus designed to optimize the usage of modern multi-core CPUs. Finally, the architecture of ARGoS is highly modular, enabling easy addition of custom features and appropriate allocation of computational resources. We assess the efficiency of ARGoS and showcase its flexibility with targeted experiments. Experimental results demonstrate that simulation run-time increases linearly with the number of robots. A 2D-dynamics simulation of 10,000 e-puck robots can be performed in 60 % of the time taken by the corresponding real-world experiment. We show how ARGoS can be extended to suit the needs of an experiment in which custom functionality is necessary to achieve sufficient simulation accuracy. ARGoS is open source software licensed under GPL3 and is downloadable free of charge.

Keywords Simulation · Swarm robotics · Multi-robot systems · High-performance · ARGoS

1 Introduction

Simulation is an indispensable tool to prototype multi-robot systems solutions. A simulator makes it possible to test ideas in a safe and controllable environment, preventing damage to the (often expensive) robot platforms and other objects. There are two key requirements for a multi-robot simulator: *flexibility* and *efficiency*. Flexibility refers to the possibility for users to add new features targeted at a particular experiment, for example, new robot types or new sensors. Efficiency, on the other hand, refers to the ability to provide satisfactory run-time performance.

No existing mainstream simulators achieve both flexibility and efficiency. Simulators that focus on performance typically achieve efficiency by targeting specific use cases, thus sacrificing flexibility. Simulators that allow flexibility, in contrast, do so at the expense of performance and their run-time typically exceeds real-time as soon as more than a few dozen robots are simulated. In this paper, we argue that the failure of existing simulators to achieve both flexibility and performance results from fundamental design limitations. In this paper, we present *ARGoS (Autonomous Robots Go Swarming)*, a simulator based around new design principles, through which we achieve flexibility and efficiency at the same time.

To achieve flexibility, we made ARGoS modular at every level. All the main architectural components are implemented as software modules selectable at run-time (i.e., *plug-ins*). Users can override or extend the existing plug-ins to fully customize the simulator for any given experiment. Plug-ins include robot control code, sensors and actuators, as well as physics engines and visualizations. Robots and other simulated objects are also plug-ins, and can be built from simple, reusable components. Multiple implementations of any plug-in are possible. Various implementations of a single plug-in typically differ in accuracy and computational cost. By choosing appropriate plug-ins, the user can allocate accuracy

G. Di Caro · F. Ducatelle · L.M. Gambardella
IDSIA, USI-SUPSI, Galleria 2, 6928 Manno-Lugano, Switzerland

G. Di Caro
e-mail: gianni@idsia.ch

F. Ducatelle
e-mail: frederick@idsia.ch

L.M. Gambardella
e-mail: luca@idsia.ch

and, thus, computational resources for specific aspects of the simulation. In this way, the simulation remains accurate where necessary, while neglecting superfluous calculations that would negatively impact speed.

Efficiency is not only achieved through appropriate allocation of computational resources. In addition, the architecture of ARGoS is parallel and designed to maximize the usage of modern multi-core CPUs.

Finally, the most distinctive feature of ARGoS is the possibility to partition the simulated space into non-overlapping sub-spaces and assign each sub-space to a separate physics engine. Physics engines can be of different types (e.g., two- or three-dimensional, kinematics/dynamics, etc.) and are executed in parallel. Robots, as they navigate the environment, migrate seamlessly from engine to engine.

We designed ARGoS during the EU-funded Swarmanoid project[1] (Dorigo et al. 2013). This project studied coordination strategies for swarms composed of three kinds of robots: (i) the foot-bot (Bonani et al. 2010), a ground-based robot that moves through a combination of wheels and tracks, (ii) the hand-bot (Bonani et al. 2009), a robot able to climb walls and manipulate objects, and (iii) the eye-bot (Roberts et al. 2007), a quad-rotor-equipped flying robot. Experiments conducted during the Swarmanoid project showcased ARGOS' flexibility and efficiency, by demonstrating its ability to deal with large scale experiments involving different robot types. In addition to the Swarmanoid robots, ARGoS currently supports the e-puck (Mondada et al. 2009). ARGoS is the official simulator of three further EU-funded projects, ASCENS,[2] H2SWARM,[3] and E-SWARM.[4] ARGoS is written in C++ and is GPL3-licensed. It currently runs under Linux and MacOSX. ARGoS can be downloaded free of charge from http://iridia.ulb.ac.be/argos.

The paper is organized as follows. In Sect. 2, we discuss existing simulator designs focusing on flexibility and efficiency. Then, in Sect. 3, we describe the design features of ARGoS. In Sect. 4, we report experiments regarding flexibility and efficiency. Finally, in Sect. 5, we conclude the paper and indicate future research directions.

2 Related work

In this section, we review a number of important multi-robot simulators and discuss their features with respect to flexibility and efficiency. We restrict our focus to mainstream simulators, those widely used by the research community, and to less known simulators whose designs relate to ARGoS'. For a broader (although slightly outdated) discussion of development tools for robotics applications, including but not limited to simulators, we suggest the survey of Kramer and Schultz (2007).

All of the simulators we discuss, ARGoS included, are *physics-based* and *discrete-time*. This means that the robots and their interactions are modeled through physics equations that are calculated over time in a synchronous, step-wise fashion.

[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/.

2.1 Flexibility

General and flexible simulator designs are a relatively recent achievement. It was only during the last decade that CPU speed and RAM size on average personal computers became sufficient to support complex architectures, while providing acceptable performance. As a result, researchers have begun to design tools that can simulate different kinds of robots in relatively general use cases. Flexible tools are desirable because they allow researchers to concentrate on their work, without the necessity to design *ad hoc* tools for each new experiment. Moreover, standardized platforms facilitate code exchange and fair algorithm comparisons.

Currently, the most successful flexible simulators are Webots (Michel 2004), USARSim (Carpin et al. 2007) and Gazebo (Koenig and Howard 2004). The engines of Webots and Gazebo are implemented using the well known open source 3D-dynamics physics library ODE.[5] USARSim is based on Unreal Engine, a commercial 3D game engine released by Epic Games.[6] Although Gazebo and USARSim can support different kinds of robots, their architecture was not designed to allow the user to easily change the underlying models, thus limiting flexibility. Webots' architecture, on the other hand, explicitly provides an interface to the underlying ODE engine which allows the user to override the way forces are calculated. For example, Webots offers a fast 2D-kinematics motion model for differential drive robots. However, flexibility is limited by the fact that it is not possible to change the implementation of sensors and actuators.

The recent MuRoSimF simulator (Friedman 2010) is designed around the concept of *tunable accuracy*. In MuRoSimF, robots are modeled as a tree in which each aspect of the robot is a node. Nodes can be added to increase the accuracy of the robot model for specific aspects. By specifying the nodes wisely, the user can assign more computational resources to the necessary aspects of the simulation, while limiting resources on unnecessary aspects. This approach is very flexible and is specifically designed to support mechanically complex robots such as humanoid robots.

2.2 Efficiency

A simulator designed for swarm robotics systems must provide acceptable performance for large numbers of robots. In this paper, we target simulations involving thousands of robots. The simulators described in Sect. 2.1 were not designed for such large numbers. Simulations conducted with Webots, USARSim and Gazebo become slower than real time when more than a few dozen robots are simulated. A complete efficiency study for MuRoSimF has not been released.

To the best of our knowledge, the only widespread simulator in the robotics community that tackles the issue of simulating thousands of robots is Stage (Vaughan 2008). However, this is achieved by imposing significant design and feature limitations. Stage is designed to simulate only differential-drive robots modeled by 2D-kinematics equations. Its sensor and actuator models neglect noise. Stage excels at simulating navigation- and sensing-based experiments. However, due to the nature of the models employed, realistic experiments involving robots gripping/pushing objects or self-assembling are not possible.

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

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

The DPRSim and DPRSim2 simulators (Ashley-Rollman et al. 2011) developed within the Claytronics project[7] employ interesting techniques that provide remarkable performance, although for a very specific use case. The DPRSim simulator is based on a custom version of the ODE physics engine that was modified to be multi-threaded. DPRSim can simulate the physics of up to hundreds of thousands of simplistic agents called *catoms* that connect and communicate to form large ensembles that, in turn, behave as a form of programmable matter. DPRSim2 is a complete redesign of DPRSim that allows simulations involving millions of catoms. In DPRSim2, the computation is distributed across the processors of a computing cluster. However, the physical fidelity in the simulation of the catoms was sacrificed in favor of a more abstract model.

3 Design features of ARGoS

In this section, we present the main design features of ARGoS, relating them to the identified requirements: flexibility and efficiency.

In Sect. 3.1, we describe the modular structure of the ARGoS architecture. This structure is designed to enhance code reuse and to ease customization. In addition, users can choose which modules to employ for their experiments. This results in the possibility to allocate accuracy and computational resources only to the necessary aspects. In this way, experiments are both accurate and fast.

In Sect. 3.2, we illustrate the most distinctive feature of ARGoS: the possibility to partition the simulated space, and assign a dedicated physics engine to each partition. This feature has a positive impact both on efficiency and on flexibility.

Finally, in Sect. 3.3, we explain how the main simulation loop is parallelized into multiple threads, and discuss the positive effects this design choice has on efficiency.

3.1 Modularity

In software design, it is common practice to decouple a complex architecture into several interacting modules. As discussed in Sect. 2.1, 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. We 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, similarly to MuRoSimF, 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 Fig. 1, we 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, we describe the features of each plug-in type in depth.

[7] http://www.cs.cmu.edu/~claytronics/.

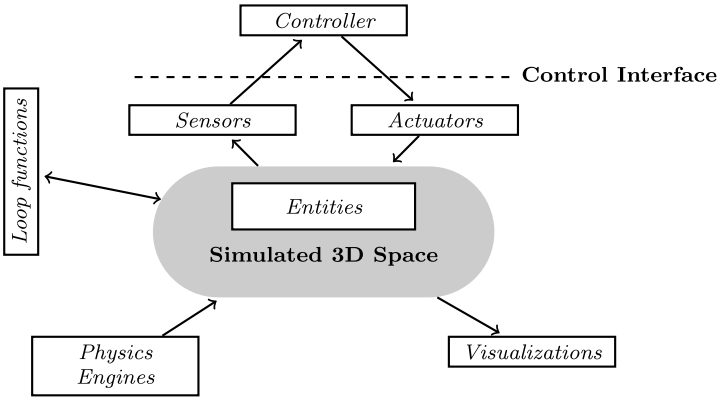


Fig. 1 The architecture of ARGoS. The white boxes correspond to user-definable plug-ins

3.1.1 The simulated 3D space

The simulated 3D space, depicted at the center of Fig. 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. For example, positional entities and their extensions are indexed in several type-specific space hashes (Teschner et al. 2003).

3.1.2 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 Sect. 3.1.1, 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.

3.1.3 Physics engines

As illustrated in Sect. 3.1.1, 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 Sect. 3.2, 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,[8] and (iv) a 2D-kinematics engine.

3.1.4 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[9] and OpenGL,[10] (ii) a high-quality rendering engine based on the ray-tracing software POV-Ray,[11] and (iii) a text-based visualization designed for interaction with data analysis programs such as Matlab.[12]

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

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

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

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

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

3.1.5 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.[13] 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.

We 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 scripting language (Magnenat et al. 2010) has already been integrated in ARGoS, and further language bindings (e.g., Lua,[14] PROTO (Bachrach et al.) 2010) are under study.

3.1.6 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, we 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 Space partitioning

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

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

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

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 partition 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, we 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, we 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 cannot 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 hash (Teschner et al. 2003). 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 Sect. 4.1, 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 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 Sect. 4.1, we empirically demonstrate that the use of multiple physics engines results in a significant decrease in simulation time.

3.3 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

```
1: Initialize
2: Visualize the simulated 3D space
3: while experiment is not finished do
4:    for all robots do
5:       Update sensor readings
6:       Execute control step                    } sense+control
7:    end for
8:    for all robots do
9:       Update robot status                     } act
10:   end for
11:   for all physics engines do
12:      Update physics                           } physics
13:   end for
14:   Visualize the simulated 3D space
15: end while
16: Cleanup
```

Fig. 2 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 Fig. 3

in Fig. 2. 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, we designed the main loop and the simulated 3D space in such a way as to avoid race conditions altogether. Our 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.

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 Fig. 2), 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 Fig. 2), 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 Sect. 3.1, 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 Fig. 2), the physics engines update the state of the *embodied entities*. Race conditions are not possible because, as explained in Sect. 3.2, 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 Sect. 4.1). A master thread coordinates the beginning and the end of each phase. In Fig. 3, 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 master thread sends the 'start' signal, the slave threads execute their work.

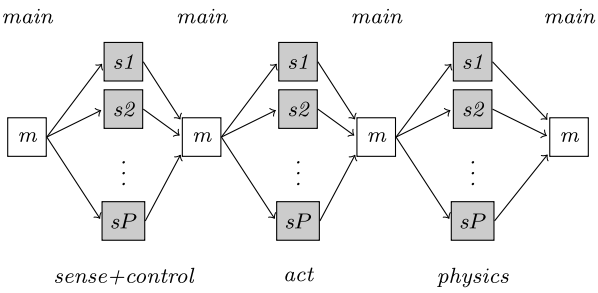


Fig. 3 The multi-threading schema of ARGoS is scatter-gather. The master thread (denoted by 'm') coordinates the activity of the slave threads (denoted by 's'). The *sense* + *control*, *act* and *physics* phases are performed by P parallel threads. P is defined by the user

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.

4 Assessment of efficiency and flexibility

In this section, we analyze ARGoS with respect to efficiency and flexibility.

In Sect. 4.1, we report the results of experiments we ran to study efficiency under several conditions. Preliminary data on ARGoS' efficiency are also reported in Pinciroli et al. (2011). To ease the analysis of the results, the experiments do not involve multiple types of robots, but rather employ a single type. We point the reader interested in experiments with ARGoS that employ different types of robots to the following papers: Ducatelle et al. (2010, 2011), Mathews et al. (2010), Montes de Oca et al. (2010), Pinciroli et al. (2009).

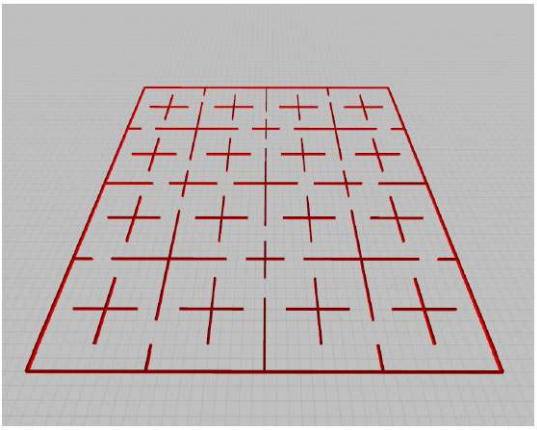


Fig. 4 A screen-shot from ARGoS showing the simulated arena created for experimental evaluation

In Sect. 4.2, we focus on flexibility. Through targeted case studies, we illustrate (i) how robots with completely different features can be composed from simple, reusable components, and (ii) how ARGoS' models can be enhanced for specific experiments to provide the necessary level of accuracy on critical aspects of the simulation.

4.1 Efficiency

As explained in Sect. 2, efficient, general-purpose simulators for multiple robots are rare. So far, little work has been devoted to characterize the efficiency of simulators for more than a few dozen robots. The only remarkable exception is offered by Stage. Vaughan (2008) 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 occlusion checks by ray casting and collision-related calculations). In addition, the robots perform a task that is meaningful for the swarm robotics community.

To test ARGoS against this benchmark, we set up an arena that mimics an indoor environment (see Fig. 4). The arena is a square whose sides are 40 m long. Similarly to Stage's benchmark, we employed the simplest wheeled robot available in ARGoS, the e-puck (Mondada et al. 2009). Each robot executes a simplified version of the diffusion algorithm proposed by Howard et al. (2002).

There are many possible ways to assess the efficiency of a simulator. In this paper, we use two metrics: (i) the time necessary to complete a simulation, and (ii) the degree of exploitation of computational resources in multi-core systems. To this end, we employ two standard measures: wall clock time and speedup. Wall clock time, in the following denoted by w, is a measure of the time elapsed between the start and end of an experiment. Wall clock time is typically affected by the quantity and type of other applications running simultaneously on the same machine. For the purposes of our analysis, we ran our experiments on dedicated machines, in which we limited the running processes to those necessary for the normal execution of the operating system and ARGoS. The second efficiency measure that we employ is the speedup, denoted by u. To obtain this measure, we 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, we obtain a measure c_i for each core. Thus, in this case, the

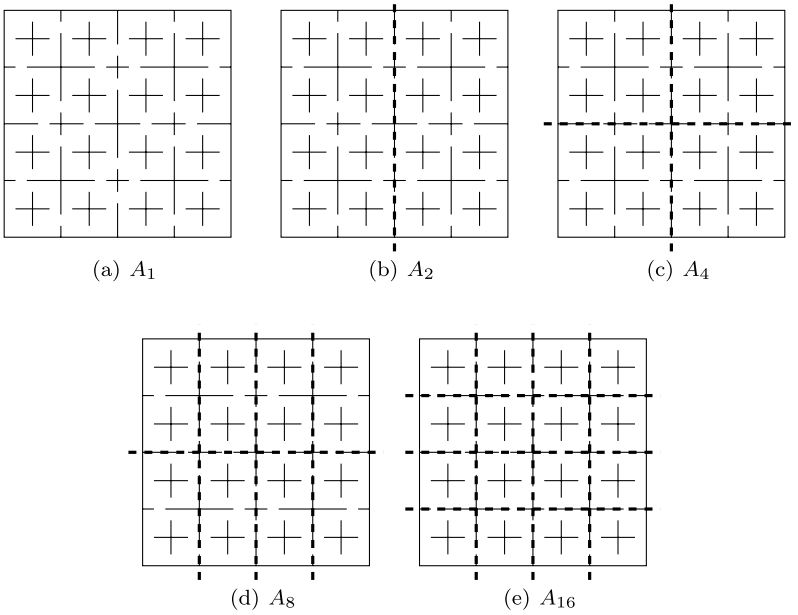


Fig. 5 The different space partitions (A_1 to A_{16}) of the environment used to evaluate ARGoS' efficiency (a screenshot is reported in Fig. 4). The *thin lines* denote the walls. The *bold dashed lines* indicate the borders of each region. Each region is updated by a dedicated instance of a physics engine

total CPU time c is given by the sum of all 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 our analysis, we focus on three factors that strongly influence efficiency: (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, we 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, we run our experiments on four machines with 16 cores each,[15] and let $P \in \{0, 2, 4, 8, 16\}$. When $P = 0$, the master thread executes all tasks without spawning the slave threads. Finally, we 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. We refer to a partition 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 partitions are depicted in Fig. 5. For each experimental setting $\langle N, P, A_E \rangle$, we run 40 experiments. The simulation time step is 100 ms long. Each experiment simulates $T = 60$ 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,

[15]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.

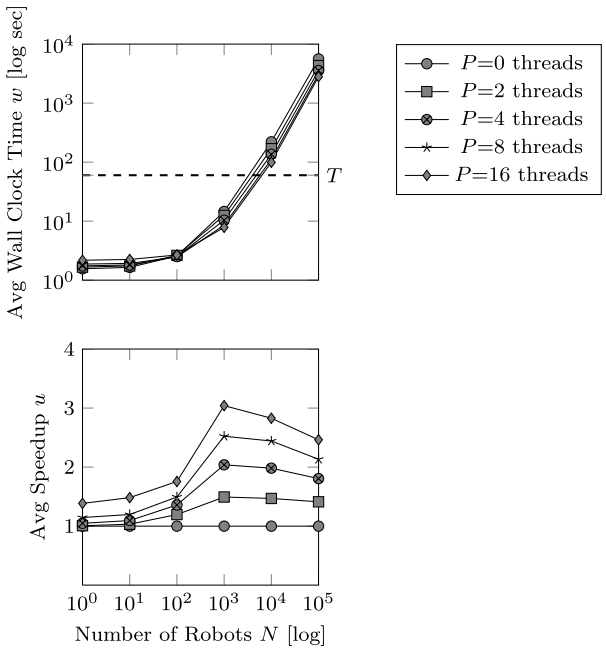


Fig. 6 Average wall clock time and speedup for a single physics engine (A_1). Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, P, A_1 \rangle$. Each experiment simulates $T = 60$ s. In the *upper plot*, points under the *dashed line* denote that the simulations 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

the measures of wall clock time and speedup include only the main simulation loop. For the same reason, we conducted our experiments without graphical visualizations.

Below, we discuss the results we obtained using different types of physics engines. In Sect. 4.1.1, we focus on the results obtained with ARGoS' 2D-dynamics engine. In Sect. 4.1.2, we discuss the result obtained with other two engines: 2D-kinematics and 3D-dynamics.

4.1.1 2D-Dynamics physics engine

In the first set of experiments, we employ ARGoS' 2D-dynamics engine. This engine is based on the efficient, state-of-the-art library Chipmunk, which is widely used in both scientific applications and games.

Single engine In the following, we report the results of experiments in which one physics engine updates all the *embodied entities* in the arena (partition A_1).

The results are reported in Fig. 6. We 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 efficiency when the number of robots is greater than 10^2. Efficiency, in terms of wall clock time, improves as the number of threads is increased. The lowest wall clock times are observed when $P = 16$. Note, for example, that when $N = 10^5$, ARGoS is twice as fast with 16 threads as it is when no threads are used.

Moreover, when threads are used, speedup is always greater than 1. The maximum speedup we observed, of approximately 3.04, corresponds to $P = 16$ and $N = 10^3$. The observation that speedup decreases in these experiments with the addition of more robots after $N = 10^3$ can be explained by the fact that only one physics engine is responsible for all the *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.[16] 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, causing speedup to decrease as the number of robots gets very large.

As the plot illustrates, the threads negatively impact wall clock time 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 thread 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.

Multiple engines Here, we discuss the results of experiments in which the arena is partitioned into multiple regions managed by different physics engines. The results are showed in Fig. 7.

A first important result is that the use of two physics engines, corresponding to space partition 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 partition A_1. Employing more threads results in increasingly better wall clock times when $N > 10^2$ and the speedup is best for $P = 16$. Comparing wall clock times with $N = 10^4$ and $P = 16$ for partition A_2 and A_1, we obtain $w(A_2)/w(A_1) \approx 0.6$.

Efficiency constantly improves with the number of engines and of threads. The most remarkable result in Fig. 7 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 thread-based parallelism increases efficiency. In our experiments, the largest speedup was observed with $P = 16$, A_{16} and $N = 10^5$.

Comparison with stage Vaughan (2008) ran Stage's efficiency evaluation on an Apple MacBook Pro, with a 2.33 GHz Intel Core 2 Duo processor and 2 GB RAM. For our evaluation, each core in the machines we utilized offers comparable features: 2 GHz speed, 1 GB RAM per thread when $P = 16$.[17]

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 whose 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.

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

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

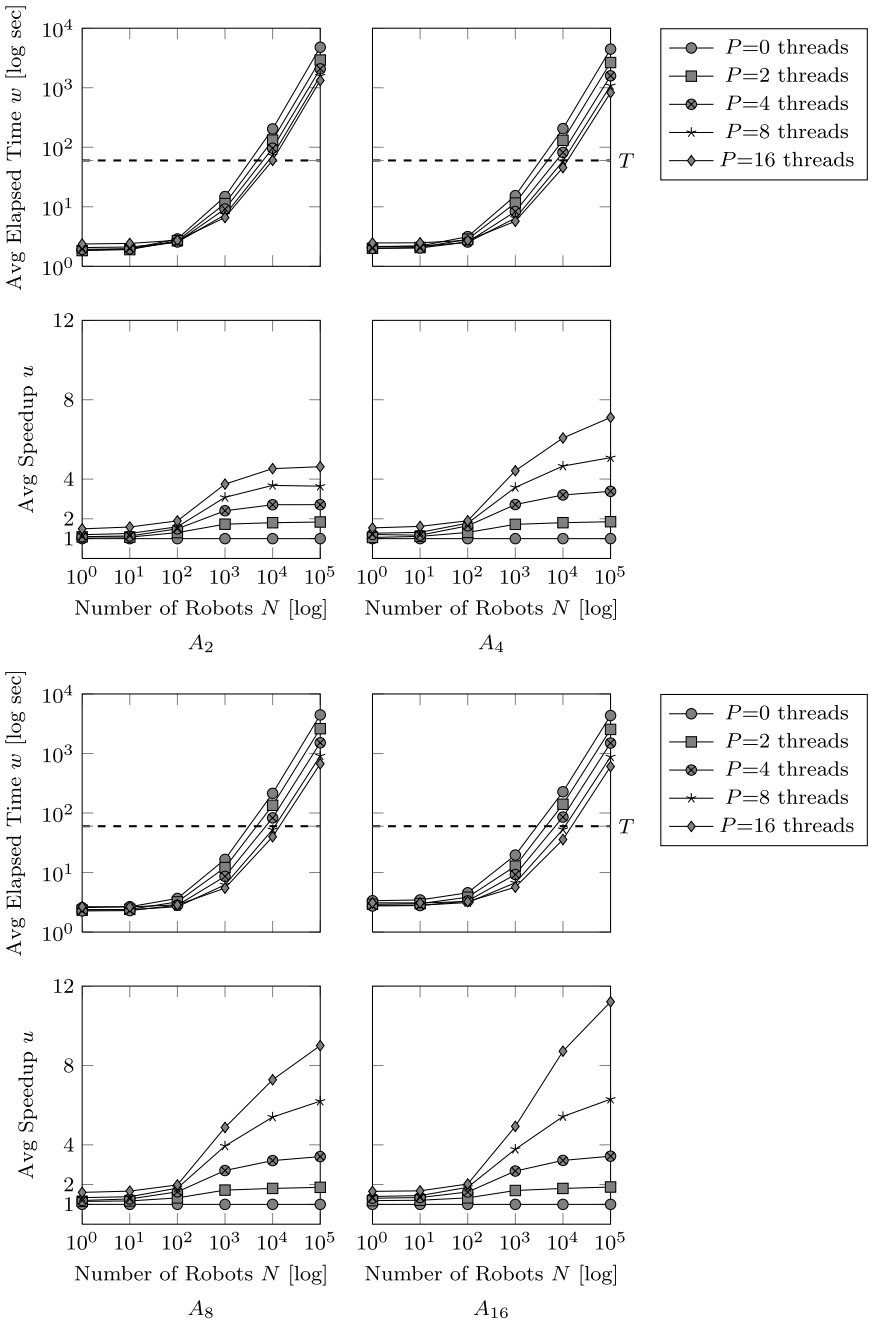


Fig. 7 Average wall clock time and speedup for partitions A_2 to A_{16}. Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, P, A_E \rangle$. Each experiment simulates $T = 60$ s. In the *upper plots*, points under the *dashed line* denote that the simulations 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

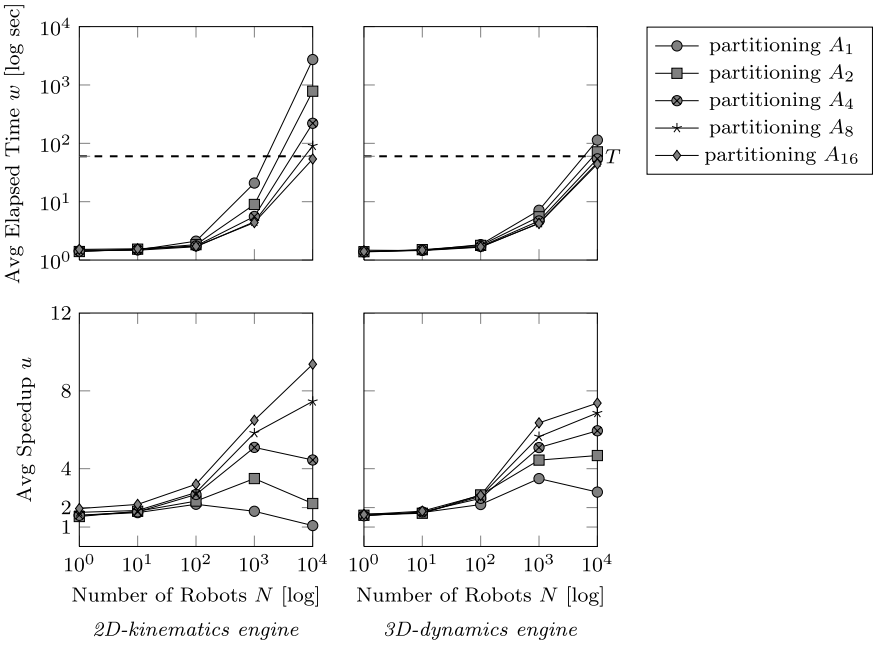


Fig. 8 Average wall clock time and speedup for experiments with 2D-kinematics engines (*left*) and 3D-dynamics engines (*right*). Each point corresponds to a set of 40 experiments with a specific configuration $\langle N, 16, A_E \rangle$. Each experiment simulates $T = 60$ s. In the *upper plots*, points under the *dashed line* denote 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

4.1.2 Results with other physics engines

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 efficiency. 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 Fig. 8 shows the average wall clock time and speedup of the benchmark experiment when the custom 2D-kinematics engine is employed. All the experiments summarized in the plot were performed with $P = 16$ threads, and with space partition A_1 to A_{16}. Results indicate that, when the space is partitioned among 16 kinematics engines, ARGoS is able to simulate $N = 10^4$ robots in approximately real-world time. Thus, even though the kinematics engine was not designed to scale, by using multiple instances of this engine it is possible to enhance efficiency to simulate thousands of robots.

3D-Dynamics engine The most detailed physics engine in ARGoS is a 3D-dynamics engine based on the ODE library. As explained in Sect. 2, this engine is used by Webots and Gazebo, two very successful robot simulators.

On the right-hand side of Fig. 8, we 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 partitions 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.

4.2 Flexibility

We illustrate ARGoS' flexibility through two case studies.

In Sect. 4.2.1, we focus on *modularity*. We explain how two completely different robots, the foot-bot and the eye-bot, are modeled into ARGoS from simple, reusable component entities.

In Sect. 4.2.2, we concentrate on *extensibility*. More specifically, we describe an experiment in which the standard models provided by ARGoS do not ensure an adequate level of simulation accuracy. Thus, we show how ARGoS can be extended with new models to fit the needs of the experimenter.

4.2.1 Modularity: the foot-bot and the eye-bot

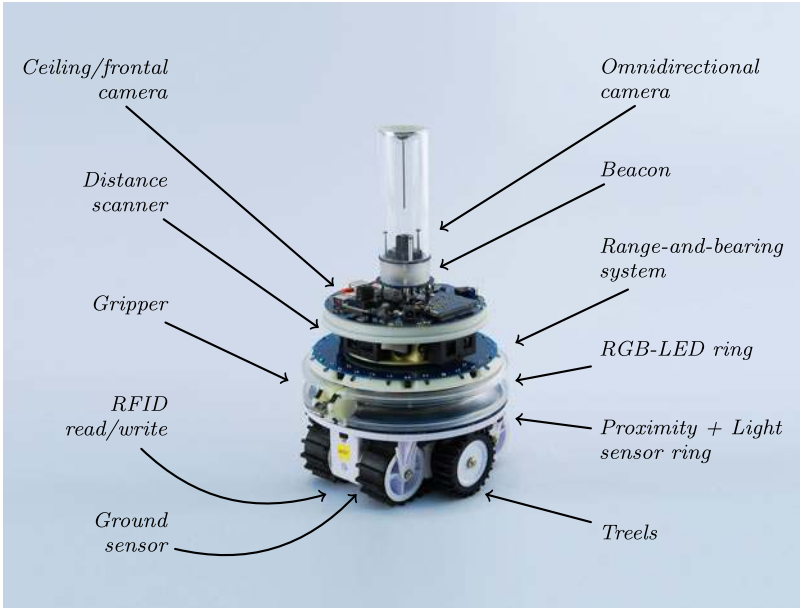
To discuss ARGoS modularity, we describe how two extremely different robots are modeled in ARGoS. The two robots we chose for this case study are the foot-bot (Bonani et al. 2010) and the eye-bot (Roberts et al. 2007). The foot-bot is a ground-based robot that moves with a combination of wheels and tracks (called *treels*). The eye-bot is a quad-rotor 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 Fig. 9.

Sensors and actuators Both robots are equipped with numerous sensors and actuators, some of which have similar features. In the following discussion, we focus on the sensors and the actuators that share common features. 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.

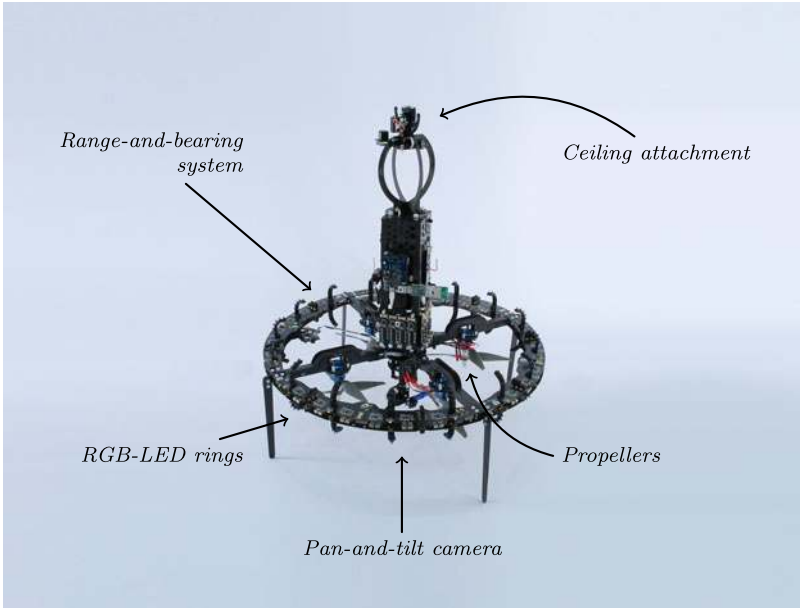
One of the simplest devices present on both robots is the on-board clock. The state of the clock is read by the clock sensor. Given its simplicity, ARGoS natively offers a single implementation of the clock sensor suitable for all robots.

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. As a result of their different configurations, the two robots have different control interfaces for their LEDs. However, thanks to the *LED-equipped entity*, the plugins for both robots can re-use the same code.

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



(a)



(b)

Fig. 9 The main devices composing two of the robots supported by ARGoS. (**a**) The foot-bot; (**b**) The eye-bot

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 a *range-and-bearing communication device* (Roberts et al. 2009), which allows 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 device is stored into a *range-and-bearing equipped entity*.

4.2.2 Extensibility: task partitioning in cooperative foraging

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

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.

Experimental setup A swarm of six 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 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. In this scenario, each robot can transport one object per trip at maximum. 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 sensory 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. In addition, the robots suffer from a systematic error caused by an asymmetry in the construction of the treel motors. As shown in Fig. 10, if both treels 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 is different across each trip and it is undetectable by the on-board motor sensors.

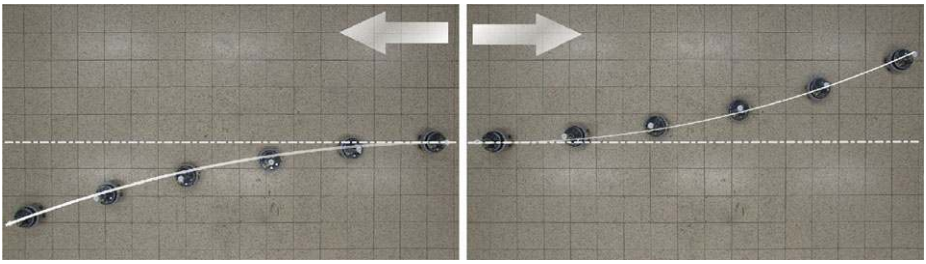


Fig. 10 When the same speed is applied to the foot-bot treels, the robot does not cover a *straight line*, due to an asymmetry in the construction of the treel motors

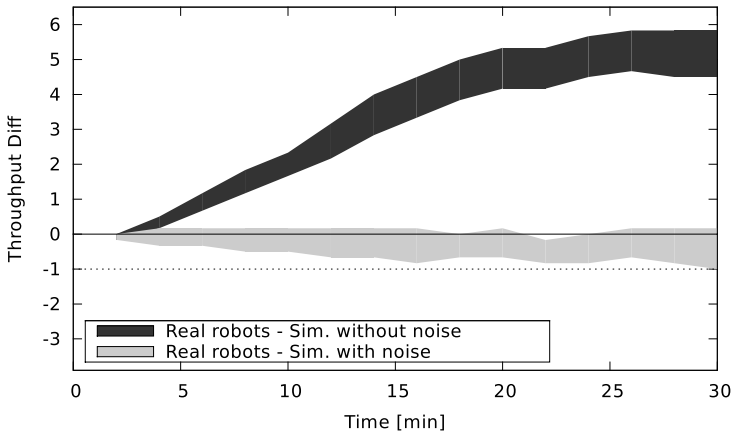


Fig. 11 Correspondence of the throughput of object transportation in simulation (with and without noise), and on real robots. The image reports the difference between simulation without noise and real robots, and simulation with noise and real robots. The *shaded areas* represent the 95 % confidence interval of the mean rank difference obtained through a Wilcoxon signed-rank test

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. (2012).

Dead-reckoning model The standard implementation of the treel motors in ARGoS does not include noise. This is a common choice in many simulators (e.g., Stage, Gazebo, Webots) because, in the vast majority of experiments, the impact of this kind of noise is negligible (see, for instance, Ferrante et al. 2010).

However, in the experiment under study, such noise plays a fundamental role on the system's performance. One of the main metrics to measure performance is 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. We consider the throughput data in three cases: simulation without noise, simulation with noise and real robots. Figure 11 shows the 95 % confidence interval on the

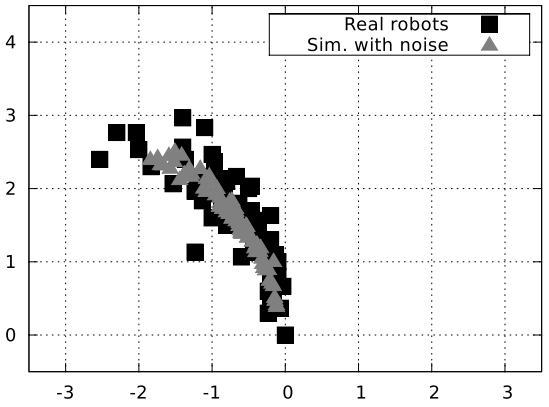


Fig. 12 Positioning error of the foot-bots with respect to their target location. We report both the data sampled from real robot experiments and from the dead-reckoning model described in Sect. 4.2.2

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. The upper bound of the confidence interval (the black shaded area in the plot) is 5.83 cm. Thus, we constructed a noise model and added a new actuator with this 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 actuator. To reproduce the slanted motion, the actuated treel speed is obtained by summing a random term to the desired treel speed set by the robot. If the motion is forwards, positive speed value, the random term is summed to the desired right treel speed; if the motion is backward, negative speed value, the random term is subtracted from the desired left treel speed. The random term is obtained by multiplying the desired speed by a term μ, taken at random from a Rayleigh distribution ($\sigma = 0.0134$). 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 12 reports the samples from real robot experiments and the data obtained with the described model.

This model is very simple and does not include a model of the treel motors. Despite its simplicity, the results illustrated in Fig. 11 show that the throughput of the robots in simulation matches the throughput in real robot experiments. The lower bound of the confidence interval on the difference between the two data sets (noisy simulation and real robot) is -1 cm (see the light gray shaded area in the plot).

Implementation in ARGoS As discussed in Sect. 3.1, 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 Sect. 3.1.6).

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 Figs. 11 and 12.

5 Conclusions and future work

We have presented ARGoS, a multi-robot simulator capable of simulating large, heterogeneous swarms of robots.

ARGoS is the first multi-robot simulator that is both flexible and efficient. Flexibility refers to the ability to simulate diverse experiments and add new features. Efficiency refers to the ability to run experiments involving thousands of robots in the shortest time possible. In state-of-the-art physics-based simulators for robotics, flexibility and efficiency are at odds. To solve this trade-off, in ARGoS we employed a set of novel design choices. ARGoS achieves both flexibility and efficiency by employing a set of novel design choices.

ARGoS' modular architecture allows the user to modify every aspect of a simulation. The user is thus able to select the most suitable modules for the experiment under study, allocating higher computational resources only to specific elements of a given experiment. Modularity enables the addition of new features, such as new robot models, sensors, or actuators, promoting exchange and cooperation among researchers.

The most unique feature of ARGoS is the possibility to divide the simulated space into non-overlapping sub-spaces, each governed by a separate physics engine. The rules implemented in each physics engine can be customized to optimize the run-time of an experiment.

Finally, the multi-threaded architecture of ARGoS exploits the resources of modern multi-core processors efficiently. Results show that ARGoS can perform accurate 2D-dynamics simulations of 10,000 robots in 60 % of real time, and accurate 3D-dynamics simulations with the same number of robots in about real time.

Future work will be devoted to improving the design of ARGoS to provide real-time performance for hundreds of thousands of robots. Possible approaches may be: (i) employing a heterogeneous threading model performing the computation both on CPU and GPU (Holmes et al. 2010) and (ii) modifying the multi-threaded architecture of ARGoS into a mixed multi-thread/multi-process architecture, in which physics engines and the simulated space are distributed across different machines in a network.

ARGoS is GPL3-licensed and can be downloaded free of charge from http://iridia.ulb.ac.be/argos.

Acknowledgements The research presented in this paper was carried out in the framework of Swarmanoid, a project funded by the Future and Emerging Technologies programme (IST-FET) of the European Commission under grant IST-022888. This work was also partially supported by the ERC Advance Grant "E-SWARM: Engineering Swarm Intelligence Systems" (grant 246939), and by the EU project ASCENS (grant 257414). Giovanni Pini acknowledges support from Université Libre de Bruxelles through the "Fonds David & Alice Van Buuren". Manuele Brambilla acknowledges support from the Fund for Industrial and Agricultural Research FRIA-FNRS of Belgium's French Community. Nithin Mathews thanks Wallonia-Brussels-International (WBI) for its support in the form of a Scholarship for Excellence grant (IN.WBI). Arne Brutschy, Rehan O'Grady, Mauro Birattari and Marco Dorigo acknowledge support from the Belgian F.R.S.-FNRS, of which they are a Research Fellow, a Postdoctoral Researcher, a Research Assistant and a Research Director, respectively.

References

Ashley-Rollman, M. P., Pillai, P., & Goodstein, M. L. (2011). Simulating multi-million-robot ensembles. In *2011 IEEE international conference on robotics and automation* (pp. 1006–1013). Piscataway: IEEE Press.

Bachrach, J., Beal, J., & McLurkin, J. (2010). Composable continuous-space programs for robotic swarms. *Neural Computing & Applications*, *19*, 825–847.

Bonani, M., Magnenat, S., Rétornaz, P., & Mondada, F. (2009). The hand-bot, a robot design for simultaneous climbing and manipulation. In *Lecture notes in computer science: Vol. 5928. Proceedings of the second international conference on intelligent robotics and applications (ICIRA 2009)* (pp. 11–22). Berlin: Springer.

Bonani, M., Longchamp, V., Magnenat, S., Rétornaz, P., Burnier, D., Roulet, G., Vaussard, F., Bleuler, H., & Mondada, F. (2010). 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)* (pp. 4187–4193). Piscataway: IEEE Press.

Carpin, S., Lewis, M., Wang, J., Balakirsky, S., & Scrapper, C. (2007). USARSim: a robot simulator for research and education. In *Proceedings of the IEEE conference on robotics and automation (ICRA)* (pp. 1400–1405). Piscataway: IEEE Press.

Dorigo, M., Floreano, D., Gambardella, L., Mondada, F., Nolfi, S., Baaboura, T., Birattari, M., Bonani, M., Brambilla, M., Brutschy, A., Burnier, D., Campo, A., Christensen, A., Decugnière, A., Di Caro, G., Ducatelle, F., Ferrante, E., Förster, A., Martinez Gonzales, J., Guzzi, J., Longchamp, V., Magnenat, S., Mathews, N., Montes de Oca, M., O'Grady, R., Pinciroli, C., Pini, G., Rétornaz, P., Roberts, J., Sperati, V., Stirling, T., Stranieri, A., Stützle, T., Trianni, V., Tuci, E., Turgut, A., & Vaussard, F. (2013) Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics and Automation Magazine* (In press).

Ducatelle, F., Di Caro, G., & Gambardella, L. (2010). Cooperative self-organization in a heterogeneous swarm robotic system. In *Proceedings of the genetic and evolutionary computation conference (GECCO)*, New York: ACM. Proceedings on CD-ROM.

Ducatelle, F., Di Caro, G., Pinciroli, C., & Gambardella, L. M. (2011). Self-organised cooperation between robotic swarms. *Swarm Intelligence*, *5*(2), 73–96.

Ferrante, E., Turgut, A. E., Mathews, N., Birattari, M., & Dorigo, M. (2010). Flocking in stationary and non-stationary environments: a novel communication strategy for heading alignment. In R. Schaefer, C. Cotta, J. Kołodziej, & G. Rudolph (Eds.), *LNCS: Vol. 6239. Parallel problem solving from nature—PPSN XI* (pp. 331–340). Berlin: Springer.

Friedman, M. (2010). *Simulation of autonomous robot teams with adaptable levels of abstraction*. PhD thesis, Technische Universität Darmstadt, Germany.

Holmes, D. W., Williams, J. R., & Tilke, P. (2010). An events based algorithm for distributing concurrent tasks on multi-core architectures. *Computer Physics Communications*, *181*(2), 341–354.

Howard, A., Matarić, M., & Sukhatme, G. (2002). 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)* (pp. 299–308). New York: Springer.

Koenig, N., & Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 2149–2154). Piscataway: IEEE Press.

Kramer, J., & Schultz, M. (2007). Development environments for autonomous mobile robots: a survey. *Autonomous Robots*, *22*(2), 101–132.

Magnenat, S., Rétornaz, P., Bonani, M., Longchamp, V., & Mondada, F. (2010). ASEBA: a modular architecture for event-based control of complex robots. *IEEE/ASME Transactions on Mechatronics*, *PP(99)*, 1–9.

Mathews, N., Christensen, A., Ferrante, E., O'Grady, R., & Dorigo, M. (2010). Establishing spatially targeted communication in a heterogeneous robot swarm. In *Proceedings of 9th international conference on autonomous agents and multiagent systems (AAMAS 2010)* (pp. 939–946). Toronto: IFAAMAS.

Michel, O. (2004). Cyberbotics Ltd.—Webots: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, *1*(1), 39–42.

Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.-C., Floreano, D., & Martinoli, A. (2009). The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions* (Vol. 1, pp. 59–65). Castelo Branco: IPCB.

Montes de Oca, M. A., Ferrante, E., Mathews, N., Birattari, M., & Dorigo, M. (2010). Opinion dynamics for decentralized decision-making in a robot swarm. In M. Dorigo et al. (Eds.), *LNCS: Vol. 6234. Proceedings of the seventh international conference on swarm intelligence (ANTS 2010)* (pp. 251–262). Berlin: Springer.

Pinciroli, C., O'Grady, R., Christensen, A., & Dorigo, M. (2009). Self-organised recruitment in a heterogeneous swarm. In *The 14th international conference on advanced robotics (ICAR 2009)* (p. 8). Proceedings on CD-ROM, paper ID 176.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Caro, G. D., Ducatelle, F., Stirling, T., Gutiérrez, A., Gambardella, L. M., & Dorigo, M. (2011). ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS 2011)* (pp. 5027–5034). Los Alamitos: IEEE Comput. Soc.

Pini, G., Brutschy, A., Scheidler, A., Dorigo, M., & Birattari, M. (2012). Task partitioning in a robot swarm: retrieving objects by transferring them directly between sequential sub-tasks. Technical report TR/IRIDIA/2012-010, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.

Roberts, J., Stirling, T., Zufferey, J., & Floreano, D. (2007). Quadrotor using minimal sensing for autonomous indoor flight. In *European micro air vehicle conference and flight competition (EMAV)*. Proceedings on CD-ROM.

Roberts, J., Stirling, T., Zufferey, J.-C., & Floreano, D. (2009). 2.5d infrared range and bearing system for collective robotics. In *IEEE/RSJ international conference on intelligent robots and systems (IROS 2009)*, Piscataway: IEEE Press.

Tanenbaum, A. S. (2001). *Modern operating systems* (2nd ed.). New Jersey: Prentice-Hall.

Teschner, M., Heidelberger, B., Mueller, M., Pomeranets, D., & Gross, M. (2003). Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the vision, modeling, and visualization conference* (pp. 47–54). Heidelberg: Aka.

Vaughan, R. (2008). Massively multi-robot simulation in Stage. *Swarm Intelligence*, *2*(2), 189–208.