Chapter 11
Get Together! Multi-robot Systems: Bio-Inspired Concepts and Deployment Challenges

Vivek Shankar Varadharajan and Giovanni Beltrame

11.1 Objectives of the Chapter

At the end of this chapter, you will:

- understand the different types of multi-robot systems,
- be aware of the task allocation problem,
- be able to point out the different types of swarm programming techniques,
- be familiar with the fundamentals of swarm programming,
- understand the real-world deployment challenges with robot swarm.

11.2 Introduction

Swarm robotics is a branch of robotics that focuses on multi-robot systems that coordinate to perform complex tasks through simple behavioral rules. Swarm robotics combines multi-robot systems with swarm intelligence (Bonabeau et al., 1999), a field that studies how complex behaviors emerge from simple and local interactions (Dorigo et al., 2021) in natural systems like schools of fish, flocks of birds and colonies of insects (see Fig. 11.1). These natural systems are of high interest because they exhibit efficiency, robustness, parallelism and adaptivity. Ant colonies are an excellent model for swarm intelligence, as ants work in parallel and use incredibly low amounts of energy to perform tasks (efficiency), the loss of several ants does not compromise the colony (robustness), and they can overcome complex environmental challenges: as an example, fire ants can form rafts with their bodies to carry the colony to safety in case of floods (adaptivity). Swarm robotics research started out as an use case to swarm intelligence on virtual and physical agents. Swarm intelligence is a property of groups of simple individuals whose collective behavior exhibit

V. S. Varadharajan (✉) · G. Beltrame
Department of Computer and Software Engineering, Polytechnique Montréal, Montreal, Canada
e-mail: vivek-shankar.varadharajan@polymtl.ca

G. Beltrame
e-mail: giovanni.beltrame@polymtl.ca

D. Herath and D. St-Onge (eds.), *Foundations of Robotics*,
https://doi.org/10.1007/978-981-19-1983-1_11

Fig. 11.1 Some examples of natural swarms are a flock of birds, colony of bees, schools of fish and swarms of ants. *Credits* Bee colony—flickr.com/Sy, Fish school—iStock.com/armiblue, Army ants—flickr.com/Axel Rouvin, Ant raft—wikimedia.org/TheCoz and Starling swarm—wikimedia.org/Walter Baxter

capabilities that are beyond the capacity of a single individual. The phenomenon of having many simple things performing complex activities when working as a group is known as *emergence*.

Swarm intelligence was initially applied to virtual agents as an approach to solve optimization problems that are otherwise considered very hard. Some examples of such computational algorithms are ant colony optimization (Dorigo et al., 2006) and particle swarm optimization (Kennedy & Eberhart, 1995). Ant colony optimization applies the foraging behavior observed in ants to optimization: a group of simulated agents move randomly in the search space (i.e., the space of possible parameters), locate optimal solutions and lay virtual pheromones (analogue to the chemical traces left by real ants) to direct other agents. Similarly, particle swarm optimization uses a group of agents moving in a search space. These techniques have been very successful in a wide range of domains like antenna design (Chang et al., 2012), vehicle routing (Bell & McMullen, 2004), and scheduling problems (Xing et al., 2010).

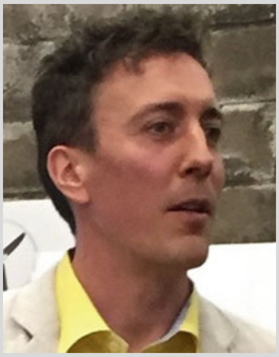
Applying swarm intelligence to multi-robot systems in the real world is not as straightforward as for virtual agent based optimization algorithms: robots need to perceive their environment, determine their position, interact with other robots and the (potentially unstructured) environment itself. Performing all these activities in a single complex robot is already a daunting challenge, and having them emerge from the interaction of many simple robots requires novel approaches to design and synthesize robotic systems. This additional complexity means that only a very limited number of works have demonstrated out-of-the-laboratory operation capability and there is no real-world application to date that directly uses swarm robotics design principles (Dorigo et al., 2021). However, swarm robotics is rapidly finding new application domains (logistics, agriculture, space exploration, and many others) in which it can provide a definite advantage, and swarm-based real-world applications are bound to happen in the near future.

In this chapter, we will provide a brief introduction to multi-robot and swarm system design approaches, swarm programming concepts and finally outline some challenges to be addressed in realizing a real-world swarm system.

An Industry Perspective

Patrick Edwards-Daugherty

Spiri Robotics



My formal education was in mathematics, applied to theoretical physics. I began programming at a young age, among my other interests in chess, music, and science fiction. As a child, I was inspired by the positive and hopeful thinking of imaginative writers and scientists. About a year after graduating in 1998, I started a tech company. I pivoted to robotics in 2012. The use of robots for space exploration had always been interesting to me from a distance. But that year, when I saw early displays of small drones able to maneuver without human control, I became convinced of a tangible possibility to create truly autonomous robots that could improve the human condition.

When my company's robotics team was at the first major public exhibition of our work, at the most embarrassing moment, our batteries caught fire in their recharging cradles. For the rest of the conference, a security guard with a fire extinguisher was stationed next to our display. He was very pleasant and supportive. In my journey with robotics, I have found the biggest cliffs are the ones right between the "completion" of a design and algorithm on the board, and the first field test that works out. As a result, at my company, we try to fail fast and often (and as much as possible, inexpensively) as part of our method. Ensemble action by autonomous agents, sometimes called swarming or flocking, first needed a basic method for group communications and consensus. The way a flock of starlings or a school of anchovy can move as one is an inspiration. The communications part has come a long way in the past decade. The next challenge, which will remain a challenge for a long time, is to figure out what actions are useful for the robotic ensembles to engage in, and what, specifically, are the desired outcomes, so these can be programmed and optimized. The communication part is the underlying first step, and each action can be thought of as analogous to a group behavior of an animal species. There are many, and they are very particular to the need and context.

11.3 Types of Multi-robot Systems

Robot swarms are a special type of multi-robot systems that rely on three guiding principles: (a) control is *decentralized* (i.e., there are no external controlling entities); (b) there are no leaders or predefined roles; and (c) robots make decisions based on local interaction with other robots. To better understand robot swarms, we must introduce a taxonomy of multi-robot systems, clarifying the differences between decentralized approaches (such as robot swarms) and other types of multi-robot systems. Multi-robot systems (MRS) are generally considered to have two or more robots that coordinate to perform a task. The robots in an MRS can be simple, as the actual potential of the system can lie within group's emergent behavior. Consider the task of collaborative transport (as seen in natural ant colonies): robots need to lift and move an object that would be too heavy for a single robot. In this case, a single robot is incapable of performing the task, but several robots can, although requiring a high degree of coordination. In general, multi-robot systems are preferred for large, spatially distributed tasks which benefit from the inherent parallelism of using multiple robots.

An MRS can be homogeneous or heterogeneous: a homogeneous MRS is composed of identical robots (same sensors, computing resources and actuators), while a heterogeneous MRS contains robots that are fundamentally different (in sensors, computing resources and/or actuators). Homogeneous MRSs are the most common type of MRS because they are relatively simple to design and manage, whereas heterogeneous MRS design needs sophisticated task planning to determine the appropriate type of robot to perform each task.

MRSs can be further classified into centralized, distributed and decentralized based on the decision making strategy that they use, as illustrated in Fig. 11.2. In Fig. 11.2: (a) centralized system with each robot connecting to a central server, the centralized server performs the decision making. (b) One of the robots is elected to perform decision making in a distributed system. (c) All the robots in a decentralized system perform decision making on-board by collecting information from other robots.

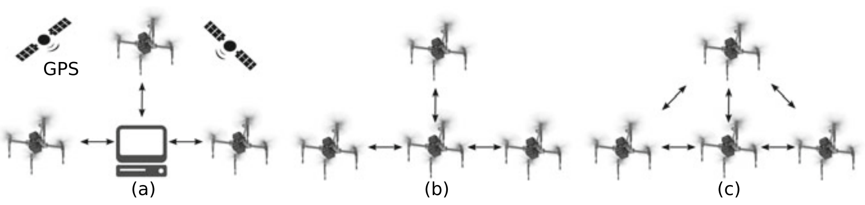


Fig. 11.2 Decision making architecture classes: **a** centralized, **b** distributed and **c** decentralized

Similar to other fields of research, decision making in an MRS can be considered as a process of analyzing a sequence of alternatives to determine the best choice of action to perform, using the available information. Most general forms of MRS design brakes down the global problem into smaller sub-problems that can be assigned to individual robots. Individual robots take up one or more of these sub-problems and work toward solving them. Task allocation (TA) is a process of optimally assigning tasks to a robot that will maximize the overall system performance and it can be considered as an example of decision making. TA in a MRS is commonly referred to as Multi-Robot Task Allocation (MRTA), where a set of tasks are assigned optimally to a set of robots to maximize the overall performance of the system.

An MRTA is generally modeled as a combinatorial optimization problem: consider N_t to be the number of sub-tasks that need to be assigned to N_r robots to minimize the global combined cost, or maximize the reward. The cost function (a metric to define the quality of global task performance) and the customized constraints on the optimization for each of the robots depend on the specific task performed by the robots. The goal of the optimization problem is to obtain a specific sub-task assignment for the robots, which is generally defined by the tuple (r_i, t_i), where $t_i \in 1, \ldots, N_t$ and $r_i \in 1, \ldots, N_r$. A generalized MRTA problem is of the following form:

$$\max \sum_{r_i=1}^{N_R} \sum_{t_i=1}^{N_T} b_{it} x_{it} \tag{11.1}$$

subject to

$$\sum_{r_i=1}^{N_R} x_{it} \leq 1 \qquad \forall t_i \in 1, \ldots, N_T$$

$$\sum_{t_i=1}^{N_T} x_{it} \leq L_T \qquad \forall r_i \in 1, \ldots, N_R$$

$$x_{it} \in \{0, 1\} \quad \forall r_i \in 1, \ldots, N_R \quad \forall t_i \in 1, \ldots, N_T$$

where b_{it} is the reward accumulated by assigning the task t_i to robot r_i. $x_{it} \in \{0, 1\}$ is a binary variable indicating whether robot r_i is assigned to task t_i. The constraint $\sum_{r_i=1}^{N_R} x_{it} \leq 1, \forall t_i \in 1, \ldots, N_T$ restricts that assignment of one single task to one robot. L_T indicates the maximum number of tasks that can be assigned to each robot; when $L_T = 1$, it is referred to as single-assignment problem, where every single robot only performs one task.

11.3.1 *Centralized Multi-robot System*

Centralized MRS generally have a single hub, either a server or a robot, which gathers the sensory data from all the robots and aggregates a global view to then perform task allocation. A centralized MRS effectively is one large system with a global view of the environment and the states of all the robots, and hence, this system has the ability to produce globally optimal task assignment and plans. There exists a wide variety of centralized decision systems (Luna & Bekris, 2011; Wurm et al., 2008; Yan et al., 2010). Some of them rely heavily on centralized localization (like Global Positioning System, GPS) and a few other approaches (McLurkin, 2009) suited for indoor applications use motion capture systems or ceiling-mounted camera. An interesting example of a centralized system is the Intel Shooting Star drones, which have been used in several light shows. These aerial robots form a large pack that operate synchronously to create 3D visual effects in the night sky. These drones have a centralized coordination stations to plan predetermined GPS trajectories and perform role-specific behaviors that are pre-scripted.

While these pre-scripted displays are impressive, the ability of centralized approaches to handle dynamic environments is limited and does not scale efficiently for larger numbers of robots. Centralized approaches also have other drawbacks—they are not robust to robot failure and are vulnerable to security threats due to their single point of failure: if the centralized hub malfunctions or compromised, the system is rendered useless.

11.3.2 *Distributed Multi-robot System*

Distributed MRS uses opportunistic centralization, where one robot in the system (referred to as the "master") is elected to act as a centralized hub that receives task-related information from all robots for TA. The term opportunistic centralization is used mainly because centralization is performed only for the time being until the TA is performed; for the next round of TA, a different robot or the same robot is used. Distributed MRS is used in a wide range of application domains, for instance, in formation control (Michael et al., 2008), exploration control (Sheng et al., 2006) and navigation control (Fan et al., 2020). Distributed MRS is comparable to a distributed computing cluster (Hwang et al., 2013). The main difference with a distributed computing cluster and a distributed MRS is that the nodes rely on a static topology with reliable communication, failures are rare (nodes operate in safe server rooms), and state of the system is completely controllable. The election of a master compute node and task assignment in a distributed MRS is generally done through auction, voting and assignment. In an assignment, the master node aggregates all the information of other nodes and makes a decision on the task assignment without any feedback from the other nodes in the system. In contrast to assignment, both auction and voting involves receiving a resource estimate (bid) or a preference for performing a certain task from each robot in the system to perform the TA.

Auction

An auction can be generally considered an activity in which a seller presents an item for sale to a set of buyers. For instance, in a distributed MRS, the auctions are for assigning a particular sub-task to a given robot in the system, with the seller being the central robot and the buyers being all other robots in the system. An auction is a preferred routine when the sellers do not have a good estimate on the buyers true value of an item. Here, we will briefly discuss the common types of auctions, and for a more detailed comparison, you can refer to Chap. 9 of Easley et al. (2012).

- English auctions: This type of auctions is also known as the ascending-bid auction. In this type of auction, the seller raises the price of an item and the bidders drop out of the auction gradually until there remains only one final buyer, who is declared as the winner.
- Dutch auction: This type of auctions is also known as the descending-bid auction. In this type of auction, the bidders gradually decrease the price of the item until one of the bidder accepts the current price. The bidder that accepts the current price is declared as the winner.
- Japanese auction: In this type of auction, the value of the item starts with a zero price. The bidders gradually increase the price, bidder leave auction when the price becomes too high and the last bidder standing is declared as the winner.
- First-price sealed-bid: In this type of auction, the bidders submit closed bids that are unknown to other bidders. The highest bidder is declared as the winner.
- Second-price sealed-bid: This type of auctions are also known as the Vickery auctions, named after the Noble price winner Willium Vickrey. In this type of auction, the bidders submit closed bids to the seller; the highest bidder is declared as the winner and will pay the second-highest bid value.

The auctions can be further classified into sequential, parallel and combinatorial based on the order the tasks are sold by the seller. In a sequential auctions, the seller sells the items one at a time until all the items are sold; the auction lasts several rounds until all the items are sold. Parallel auction requires the seller to sell all the items at once and the buyers bid on it in parallel; the auction of all the items is performed in one single round. In a combinatorial auction, the seller sells a combination of different items, and the bids are cast on packages of items; the seller sells the items based on an assignment that maximize the revenue.

In a MRS, the bids are generally determined by the sellers' cost (resources required) in performing the task. For example, if the tasks (item) correspond to spatial goals the robots have to reach, the cost of reaching the goal (distance) is fixed. First price sealed bid is one of the most commonly used type of auction in MRS (Otte et al., 2020), mainly because of the nature of the tasks involved and the auctions can be performed in a single round rather than multiple rounds (as in English, Dutch and Japanese auction). However, there are type of tasks that are favorable for multi-round auctions that use other type of auctions. We refer the reader to Otte et al. (2020) for more information on types of auctions that are used in MRS for different types of tasks.

Voting

The voting is generally considered to be an activity in which a group of individuals express their preference over a sequence of alternatives which are aggregated to obtain the preference of the whole group. Voting and auctions are both used to aggregate information across a group; thus, it might be hard to completely distinguish between the two. However, the circumstances under which voting and auction are used can be clearly distinguished. Voting is applied when the group is trying to reach a single distinct decision that defines the group preference using individual preferences, whereas auction is applied when an estimate on the choices (bids) can be used to aggregate the preferences.

In voting, there exists a set of alternatives m that needs be ranked by each individual as strictly dominating $A \prec_i B$ or as weakly dominating $A \preceq_i B$, with A and B being the two alternatives. $A \prec_i B$ means that individual i strictly prefers alternative A over alternative B and $A \preceq_i B$ means A is preferred weekly by individual i over B. With these individual preferences in hand, different types of rules are applied in various voting systems to aggregate the individual preference.

- Plurality: This type of voting is also called the majority rule and considered the most natural way of voting on alternatives. In this type of voting, each alternative receives a score when it is ranked first and the group preference is a ranking that is produced with the aggregated score.
- Borda Count: An alternative receives a score of m when it is ranked first by an individual, receives m-1 when it is ranked second and 0 when it is ranked last. A summation of all the individual ranking scores are produced to obtain the aggregated group preference.
- Copeland Count: Elections are conducted pairwise between individuals; the alternatives that win a pairwise election receive a score of two, a score of one for a tie and no points for a defeat. An aggregated score of all the alternatives are produced to obtain the group ranking. The alternative that wins the most pairwise election is ranked first.
- Bucklin: The alternative that receives a first ranking from more than half of the voters is placed as the first group ranking alternative; if there is no alternative that is preferred first, then the second alternative preferred by more than half of the voters is ranked first. This process of selecting the alternative is iteratively performed to obtain the group ranking.
- STV: The alternative that receives the least votes is removed in each round of voting and the last standing alternative is ranked first.
- Slater: A combined ranking is produced with alternatives that is consistent with the majority of the pairwise elections.
- Kemeny: A group ranking of alternatives is produced based on as few disagreements as possible. For each disagreement, an alternative is pushed behind in ranking, and a final ranking is produced with this final disagreement ranking of alternatives.

These voting procedures are reasonable and produce desirable properties, but it might be difficult to clearly distinguish the advantages between these procedures (Kacprzyk et al., 2020). A well founded set of evaluation criterion might be required to evaluate the different advantages of these systems. In the context of MRS, plurality or majority count is commonly used, since it is the most natural form of voting and applies to the type of tasks dealt in MRS (Karpov et al., 2016). There are some application scenarios like formation selection (Iocchi et al., 2003), where voting procedures like Bucklin is applied to select a formation that is preferred by more than half of robots.

Many of these voting systems produce different group preferences based on the order the voting is conducted; this gives rise to two important properties: Unanimity and independence of irrelevant alternatives (IIA). Unanimity states that when $A \prec_i B$ is the preference of every individual in the group, then the group ranking should reflect this preference. Whereas, IIA requires that the group ranking between A and B should only depend on the individual preferences between A and B and not on the preference of other alternatives. Using these two properties, we can now state Arrow's impossibility theorem: *A voting system that satisfies both unanimity and IIA must correspond to a dictatorship by one individual, when there are three alternatives or more.*

Arrows impossibility theorem essentially means the voting system that satisfies both Unanimity and IIA will not suffer from the drawback of the order in which voting is conducted. However, there is no voting system that will satisfy both these properties for more than two alternatives.

11.3.3 Decentralized Multi-robot System

Robot swarms being a subset of decentralized multi-robot systems, arise from the intersection of two domains: collective robots and swarm intelligence.

The key design principle that is followed in the design of decentralized systems are:

- Control should be decentralized: All robots in the system are considered to equip independent decision making capability.
- No leaders: There should be no master node that coordinates and manages the agents in the system.
- No predefined agent roles: There should be no fixed role for agents in the system.
- Simple, local interactions: All the interactions with the agents should be simple and should happen only on a local scale (within the communication range).

These rules directly apply to a concept generally referred to as *emergence*. Emergence is a property that a system exhibits which the individual parts of the system are incapable of exhibiting on their own, a behavior that demonstrate emergence is called emergent behavior. In the context of multi-robot systems, emergent behavior can be thought of as collective behavior that is exhibited by the system as a whole when they aggregate together. These kind of collective behaviors are widely found in natural swarms. Consider, a school of fish that exhibits a circling behavior as a measure to protect itself from predators (Fig. 11.1 shows one such circling behavior). The system that demonstrates emergence is in general very attractive because they exhibit some inherent capability to produce the following properties: scalability, efficiency, robustness, parallelism and adaptivity. Swarm robotics is the field of engineering that study emergence in robots.

Swarm robotics design problem

The problem of the design of swarm systems (see Fig. 11.3) can be defined as: given a set of high-level requirements for a swarm, how can these requirements be translated into a set of robot rules. For instance, consider the task of cleaning a room, the high-level requirements is cleaning the room and the designer task is to derive a set of robot commands to satisfy the requirements. In formal terms, the design problem is to drive the states of all the robots from an initial state to a desired final state. Consider the state of the swarm $S_0 = \{s_1, s_2, \ldots, s_n\}$, with s_i being the initial state of robot i; the goal is to derive a function $f : S_0 \rightarrow S_T$ that will drive the system to a desired final state $S_T = \{s_{1,T}, s_{2,T}, \ldots, s_{n,T}\}$ within the swarm state space $\mathbb{S}$. Before we delve into the methods available to design these rules, it might be useful to first understand what are the states of the system and how could one model a swarm system.

Swarm states

The swarm state space ($\mathbb{S}$) contains all the possible configurations for all the robots in the swarm, and each of these configurations is called a swarm state, i.e., a combination of all the individual robot states. Each individual robot has a different perception of the environment, communicates with different neighbors and hence has a potentially unique internal state. Figure 11.4 shows the individual robot states that are combined to form the overall swarm state. The individual robot state in the swarm can be broken down into:

- environmental state s_e: the state of the robot surroundings,

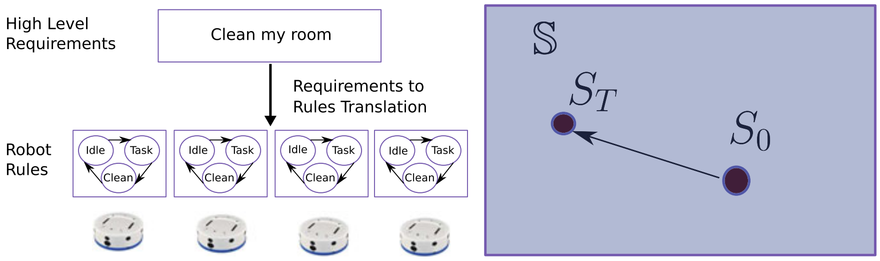


Fig. 11.3 The swarm robotics design problem: how do we translate swarm-level instructions to commands for each robot? More formally, how do we change the system state $S_0 \in \mathbb{S}$ to a desired S_T?

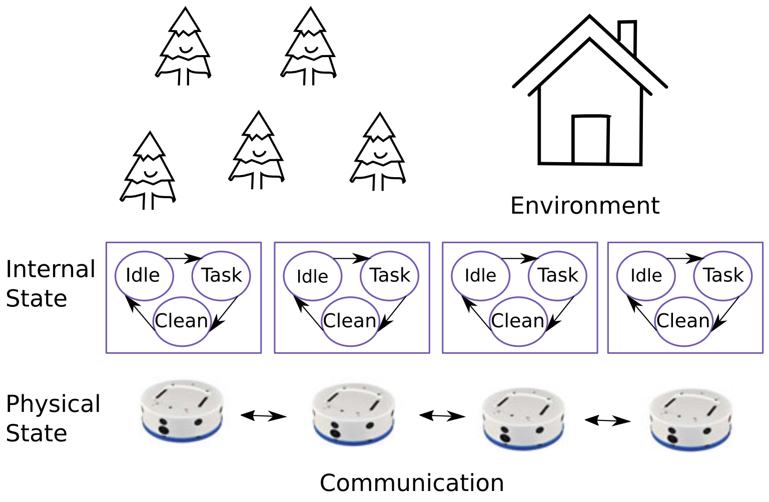


Fig. 11.4 States of the robots in a swarm are a combination of the environmental state, internal state, physical state and communication state of the robot

- internal state s_i: e.g., battery level, memory use, etc.,
- physical state s_p: the state of the sensors, actuators, and other mechanical parts of the robot,
- communication state s_c: the internal state resulting from communication with neighbors.

Consider, $s_\text{i} = \{s_\text{e}, s_\text{i}, s_\text{p}, s_\text{c}\}$ the state of the robot i, the state of the swarm would be $S = \{s_i\}, i \in (1, n)\}$ with n the number of robots in the swarm. As one can observe, the swarm state contains the state of all robots, making the formal modeling of swarms rather challenging.

A swarm of robots is generally considered as a single machine with evolving state, and it is generally called an open machine, since only a parts of its state is controllable. The environmental state s_e around the robot is dynamic, and it can only be partially modeled because the sensors of the robot are only capable of capturing a subset of the environmental state with some amount of uncertainty. Furthermore, the environment around the robot keeps evolving as the robot is performing its task and can only be considered partially controllable. Similarly, the physical state s_p of the robot is also only partially controllable (e.g., the battery level cannot be controlled). Another reason for s_p being partially controllable is due to the presence of a non-zero probability for a hardware failure. On the contrary, the internal state of the robot s_i is considered to be fully controllable by the robots through programming.

The communication state s_c depends on the underlying communication topology created by the robots and the state of the communication medium. When more and more robots are sending information, the chances of collisions and packet drops increase. Every robot in a swarm is assumed to have a limited communication range

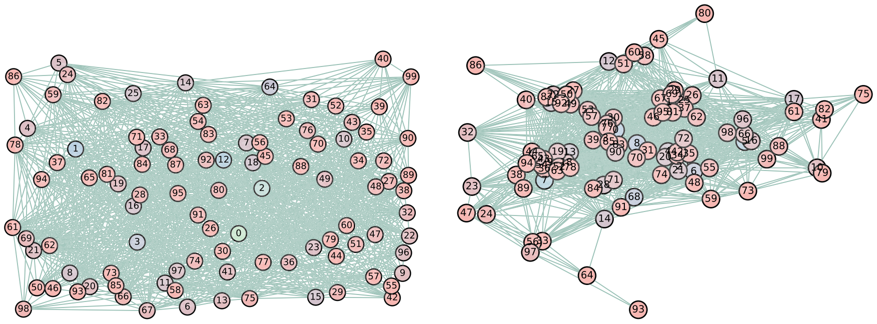


Fig. 11.5 Communication topology created by a swarm of 100 robots: on the left the robots form a cluster topology and on the right a scale-free topology

based on its underlying communication hardware. The ability of the robots to communicate can be represented by a graph structure called the communication topology. In a communication topology graph, the nodes represent the robots and the edges represent the communication links between them. Figure 11.5 shows some types of communication topology, scale-free, and cluster topology. The communication topology of a MRS is continuously affected by the movement of the robots, and the communication affects movements; hence, only a part of this state is controllable. The problem of connectivity maintenance (i.e., maintaining a desired communication topology) is usually formulated as a dual problem that addresses both movement and communication simultaneously.

Design of swarm robotic systems

The task of programming a swarm robotic system starts by the definition of the requirements that define what the swarm is required to accomplish. The requirements are then translated into a set of robot rules defining the behavior of each of the robot in the swarm. The task of the programmer is to create these robot rules from the requirements, which can be generally referred to as the control software design process. The design of control software for robot swarms can be manual or automatic.

Automatic methods

The task of control software design is formulated as an optimization problem where the parameters for optimal robot behavior are found via search (see Fig. 11.6). The search in automatic methods generally involves a robot simulator and a set of template alternative robot control architectures. The performance of a given alternative on the swarm is then evaluated using a performance metric. The performance metric ideally captures the efficiency and effectiveness of the swarm when completing the given task. The configuration space that contains all possible alternatives to the template control architecture is referred to as the design space. The candidate solutions from the design space are drawn with some search rules to identify the optimal solution that maximize the performance metric.

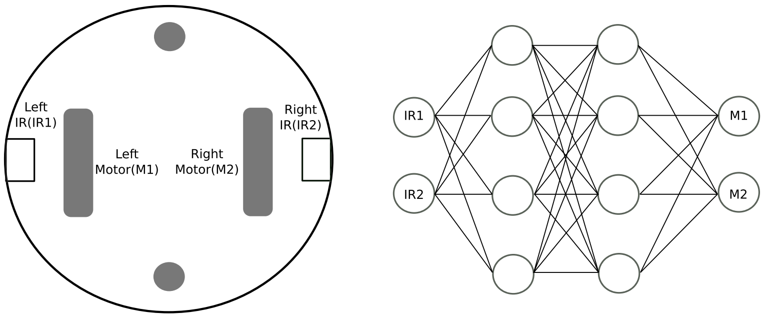


Fig. 11.6 An illustration of automatic design in its basic form, the sensory input of the robots are considered to be two binary IR sensors (IR1 and IR2), and the actuators of the robot are two motors (M1 and M2) taking real value inputs. In this scenario, an evolutionary algorithm could be used to obtain the weights of the neural network that performs the task of aggregation (similar the behavior observed in ants while forming rafts, as seen in Fig. 11.1)

The combinatorial nature of the design space demands the use of metaheuristics like evolutionary algorithm to search the design space using the performance metric as an objective function to evaluate the quality of the solutions. The most common form of automatic design is to use an evolutionary algorithm (EA) to search the design space paired with an artificial neural network (see Chap. 15 for more information on artificial neural networks) to act as the template control architecture. EA is a population-based optimization procedure inspired from biological evolution. In EA metaheuristics, a virtual population containing best performing candidate solutions are maintained; in each round of optimization, two candidate solutions are selected to be combined and produce an offspring (via procedures like crossover or recombination), and this new offspring is mutated to introduce some variance and novelty in the population. The artificial neural network maps the sensory inputs to actuation commands, and the design space contains all possible combinations of neural network weights. Some initial studies (Nol & Floreano, 2000) have proven that automatic control software design is a viable option for design of decentralized robotic systems.

Some other types of automatic design focus on designing methods that promote and search for novel behaviors in the design space, such as novelty search (Gomes et al., 2013). Novelty search promotes diversity over performance: this type of automatic design procedures are known to not suffer from problems like immature convergence or stagnation of solutions around local minima. Apart from using a neural network, automatic methods can be applied to other control architectures like parametric finite state automate (Hecker et al., 2012) or behavioral trees (Kuckling et al., 2018). Based on the type of system used to evaluate the candidate solutions, automatic methods can be further classified as online and offline.

Offline methods

The offline design process involves generating the control software before the deployment of the swarm. During the design phase, simulation is typically used to evaluating a large number of possible settings from the design space and generate and appropriate control software. The use of simulation offers the benefit of being faster than real robot evaluations and avoids damage to the physical hardware caused by low-quality candidate solutions. The most common characteristics of offline methods are:

- The behaviors produced are usually for homogeneous swarms, executing an identical version of the control software.
- The objective function (also known as the performance metric) is evaluated in a centralized manner for the whole swarm rather than evaluating the performance of the individual robots.
- A typical evolutionary approach evaluates the populations of up to 200 robots using the control software settings altered through evolutionary procedures (elitism, recombination and mutation).
- The general performance metric used is based on the spatial change of the robots relative to other robots, with an evaluation across 10–30 runs to take unknown stochastic variables into account.

Online methods

Online methods perform directly on the real deployment environment, and the performance is evaluated directly on the physical hardware. The most natural benefit of using online methods is that they can benefit from the feedback received by deploying the robots directly in the operational environment. Online methods generally produce mission specific control software rather than generalized control software that can be applied to a wide range of missions. The optimization being performed on deployed robots, only a limited number of alternatives can be evaluated due to resource limitations, and potentially robot harmful behavior has to be filtered out before evaluation. In addition, the optimization has to be distributed (with opportunistic centralization), since the swarm cannot rely on a centralized node to compute the performance metrics and guide the design space search. The limitation of using a performance indicator that can only be evaluated in a local and distributed manner makes online approaches less effective in comparison to offline design methods. However, use of hybrid approaches that combine online and offline methods is an effective way of reaping the benefit of both worlds and is under active research by the community. Some notable characteristics of online design methods are:

- The robots are asynchronously used to explore a portion of the design space, with each robot evaluating a sub-population of the evolutionary instance of the control software.
- the robots continuously exchange the best performing instance of the control software allowing other robots to include this information in its local population for further search.

- The behaviors executed by the robots are usually heterogeneous in nature; each robot executes a different control software instance. However, there is a possibility that the robots will eventually reach a point in the search, where they execute a similar version of the control software.
- As the robots are completely decentralized, the performance metric used has to be computed locally, using the information available on the robots. This severely limits the type of tasks that can apply online approaches.

Manual methods
Manual methods involve the design of the control software for the robots either by hand using a trial-and-error approach, or using the designer's expertise. The general procedure is to use a state machine to model and encode the robot control software. The state machine allows the robots to decompose the overall goal into elementary tasks. Some state transitions are performed by the swarm as a whole to ensure consensus among the individual robots in the swarm.

The designer picks a tool that best fits the task at hand and devises a set of rules that will allow the robots in a swarm to produce a self-organizing behavior. Some notable self-organizing behaviors are aggregation, circling and pattern formation. The main advantage of using manual methods is that the programmer has complete control over the design software and can customize them to best fit the robotic mission. One of the downsides of this approach is that it is very hard to manually design a decentralized behavior for the robots since only a part of the state is controllable and known to the programmer.

11.4 Swarm Programming

Swarm programming is the process of writing code to describe swarm behaviors. A swarm programming language is a *domain-specific language* that can be used for describing control software for robot swarms. Like other domains in computer science, a swarm programming language can be compiled into machine code containing a set of instructions that can be executed by each robot. The basic requirements of a swarm programming language are to provide a rich feature to allow arbitrary missions and to provide support for most robotic hardware. Other desirable properties of a swarm programming language are:

1. Composability: The control software should be able to work in parts and as well as, work as a single control software, when various parts of the code are put together. For instance, a programmer can design a particular part of the code separately as a function and test it, when putting together such similar functions, it should work as a whole behavior.
2. Predictability: When looking at a piece of code, the designer should be able to reason the behavioral outcome that will be observed on the robots.

3. Heterogeneous hardware support: The programming language should provide support for designing swarms that contain various types of robotic hardware.
4. Hardware agnostic: The programming language should produce invariable behavioral outcome across various robotic hardware. A given piece of control software designed using the programming language should be compatible to be deployed on a wide range of robotic hardware.

11.4.1 Swarm Programming Languages

Over a decade of research in the field of swarm robotics have produced a wide variety of methods that are used in programming the control software for robot swarms. In this section, we will discuss some of the notable programming languages and paradigms that are used in design of robot swarms.

Robot oriented
The main focus in robot-oriented programming is to provide the designer with as precise control as possible to program every single robot in the group. In robot-oriented programming, the designer focuses on designing an individual robot behavior that will work synchronously to realize a desired group behavior, this type of swarm programming is also known as bottom-up approach. One of the most common tool used for robot-oriented programming is the robot operating system (ROS) (Quigley et al., 2009). ROS is considered to be one of the widely used tools in programming both single robot and multi-robot systems. ROS being programming language flexible allows a designer to design a control software using various programming languages (Python, C/C++ or Java). Chap. 5 introduces the fundamentals of ROS and can be used as a reference to ROS. One of the main advantages of using ROS is the availability of several robustified packages and drivers that can be readily used to program every single robot in the group. On the downside of programming swarms with ROS, the programmer has to take into account each of the ROS node interactions and its details (not limited to the naming used to connect ROS nodes). The complexity of managing the ROS specific details increases exponentially with the number of robots in the system.

Spatial computing
Spatial computing focuses on providing programming tools for programming the swarm as a whole rather than considering the individual robot's behavior. Spatial computing can be used when the programmer is not interested in programming individual robots but would like to design group level behavior; this kind of approach is called the top-down approach. The robots in spatial computing are considered to be a collection of communicating compute devices that are distributed in an arbitrary operational space, capable of performing a local computational task. The frameworks in spatial computing abstract the individual robot and provide swarm specific primitives that will allow design of global behaviors. Some examples of spatial computing are

Proto (Beal & Bachrach, 2006) and Protelis (Pianini et al., 2015). In proto, the robots are assumed to be deployed on a manifold of space called amorphous medium with a physical and computational state. The robot program defines the way they interact with the neighbors and the environment to perform a location specific behavior in the amorphous medium. Spatial computing being a powerful tool for designing swarm behaviors still lose the robot individuality and the capability to program each robot in the swarm. Programming of heterogeneous robots with spatial computing is not possible.

Goal oriented and task oriented
Goal-oriented programming is considered a bottom-up programming approach where the individual robots are assigned spatial goals. The global task is broken down into elementary spatial goals and assigned to robots; the robots coordinate and reach these spatial goals in parallel. The main focus in goal-oriented programmer is placed on decomposing the global requirements into spatial goal rather than the logic used to perform the task. Some example of goal-oriented programming languages are SWARMORPH (O'Grady et al., 2012) and Termes (Petersen et al., 2011). Goal-oriented programming is more suited when the mission requires spatial organization among the robots and has minimal to no robot failures, since the approaches do not have contingent mechanisms for robot failures.

In task-oriented programming, the global task is broken down into a set of sub-tasks (such as spatial goals) that can be performed by a single robot and optimally assigned to robots. The robotic swarm is considered as a system with parallel machines that can be scheduled jobs using a scheduler (a system that assigns resource to a specific task). These type of systems are referred to as deterministic parallel machines in sequencing and scheduling theory (Pinedo, 2012). The task of control software design in goal-oriented programming is to formulate the global problem as a scheduling problem and design a scheduling system that will assign jobs (sub-goals) to the robots in a swarm. Task-oriented programming is considered to be a bottom-up approach since the individual robots are assigned tasks separately. Karma (Dantu et al., 2011) and Voltron (Mottola et al., 2014) are some examples to task-oriented programming. Task-oriented oriented programming is more suited for missions that can be decomposed into a set of sub-tasks that can be performed on a single robot. Task-oriented programming cannot be used in missions that require active inter-robot coordination (e.g., when a sub-task require two or more robots to complete it).

11.4.2 Programming in Buzz

Buzz (Pinciroli & Beltrame, 2016) is considered to be a hybrid domain-specific programming language that provide programming primitives similar to robot-oriented programming, spatial computing and goal-oriented programming. It allows programmers to maintain desirable levels of abstraction while programming, the swarm can be programmed as a whole (top-down) or individual robot behaviors can be designed (bottom-up) at the same time in a single control software. For instance, the language provides support for both setting the actuation commands (bottom-up) and support for neighbors management to consider the swarm as a whole and perform operations in the neighborhoods (top-down). A pure bottom-up approach suffers from scalability issues and conversely; a top-down approach suffers from inability to fine-tune individual robot behaviors. A concurrent design used in Buzz allows the designer to pick the right amount of abstraction required at the various stages of the mission.

Buzz satisfies most of the desirable properties of a swarm programming language: the code can be organized as functions and classes (composible), language syntax is intuitive with similarities to Python and Lua (predictable), swarm programming constructs allows concurrent use of heterogeneous robots in a swarm (heterogeneous hardware support) and unified Buzz virtual machine (BVM) for use with various hardware platform (hardware agnostic). These properties make Buzz a promising approach to design control software for robot swarms and hence, in this chapter, we will provide a detailed introduction to programming in Buzz.

Communication and execution model

The reference communication model used in Buzz is situated communication; it is a communication paradigm introduced by Stroy et al. (2001) and commonly used in swarm robotics. In situated communication, the receiver of a message knows the positional information (distance and bearing) of the sender using a specialized communication device. The robots using Buzz either equip such a communication device or simulate situated communication through other sensory measurements. The measurements in a situated communication device are obtained as a positional and payload pair. As illustrated in Fig. 11.7 left, the positional data includes the relative range (distance) and bearing (angle) of the sender in the receivers' coordinate frame. The payload part of the message includes a serialized messages from the internal behavior programmed on the robots. The robots in a swarm, broadcast messages, the robots in communication range receive these messages (often assumed to be line-of-sight, a requirement for situated communication devices) and process these messages. The information flow in the swarm happens in a gossip based communication (i.e., from one neighborhood of robots to another) until all the robots in the swarm have similar information.

The execution of the control software follows a discretized step wise execution phase with each step denoting one control loop (illustrated in Fig. 11.7 right). During each control loop, the robots perform the following actions in order: reading the sensors, processing the input messages, performing a loop of the code, sending messages and updating the actuation commands.

Buzz Virtual Machine

Buzz considers the swarm as a collection of devices that uses a virtual machine called Buzz Virtual Machine[1] (BVM). The BVM contains an interpreter[2] to execute the control software designed for the robots (a script called Buzz script). BVM is

[1] https://github.com/buzz-lang/Buzz.

[2] A program used to execute code, for example, Python interpreter.

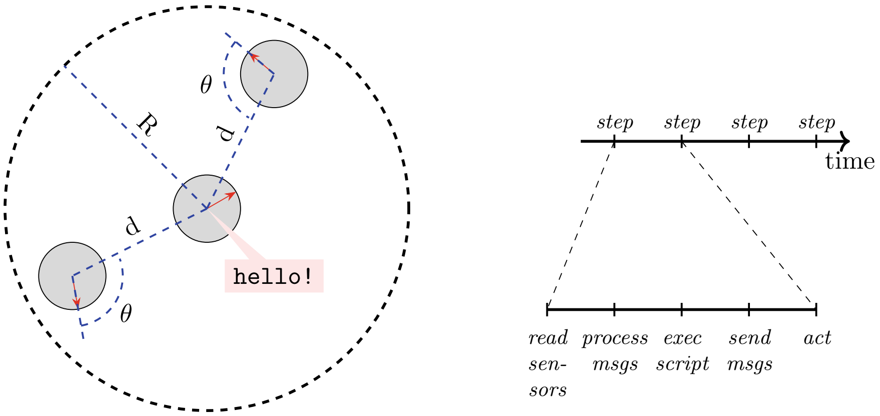


Fig. 11.7 On the left, the reference communication model performing situated communication, the sender robot in the center broadcast messages within its communication range (R) and the two receiver robots on the top/bottom, measure the distance (d) and angle (θ) of the sender in their coordinate frame. On the right, the reference execution model containing the discretized step-wise execution

written entirely in C and uses a stack (datatype providing a collection of elements)-based operations to execute the control software. Figure 11.8 illustrates the internal structure of the BVM. For more details on the BVM, we refer the reader to Pinciroli and Beltrame (2016). BVM is designed to be compact in size (about 12 kB) providing the possibility to deploy it on most of the robots used in swarm robotics; there exists a compact BVM optimized for microcontroller called BittyBuzz.[3] The internal datatype used to store information inside the BVM is key hashable tables (referred to as data holders). The reference execution architecture discussed earlier directly translates into BVM operations performed at each step: latest sensor readings and input messages update the respective data holders, the values from the data holders are used to perform a code step resulting in updating the data holders and the values from the data holders are used to update the actuation commands and output messages.

In practice, a designer writes his code in Buzz, which gets compiled into a bytecode and the bytecode is executed by the BVM. Buzz offers command line tools like *bzzc* to compile the buzz script into a BVM interpretable buzz code. The compilation is generally performed on the programmers machine and the corresponding byte code is then uploaded onto the robots for execution. Buzz is an extensible language allowing programmers to attach custom C/C++ functions as closures that can be called from the Buzz script. For instance, consider, the *take_off* routine that can be implemented for flying robots and *set_wheels* for ground robots. In the compilation phase, these custom closures are set as symbolic references and referenced during execution phase.

[3] https://github.com/buzz-lang/BittyBuzz.

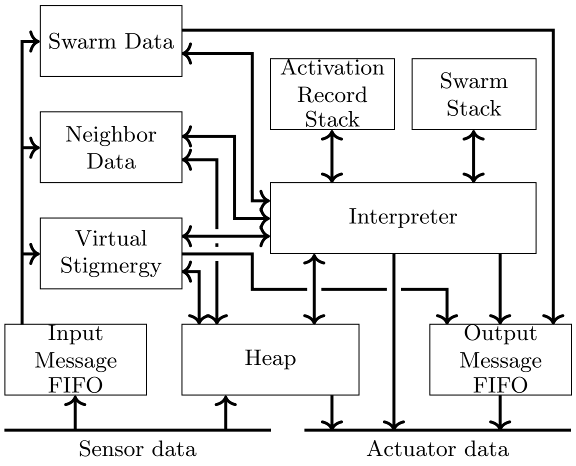


Fig. 11.8 Internal structure of the buzz virtual machine, figure obtained from Pinciroli and Beltrame (2016)

Deploying Buzz on robots

Deployment of Buzz on robots requires an adapter called the Buzz controller. The main purpose of a Buzz controller is to connect the robot sensors and actuators to the BVM data holders that store sensor and actuator information. Buzz controller also serves the purpose of connecting the communication hardware with the BVM, updating the in and out message queue inside the BVM. Buzz controllers are comparable to a hardware abstraction layer (HAL) that abstracts the robot specific sensor/actuator communication to the BVM. There exists several buzz controllers that can be readily used for robot deployments: 1. ARGoS Buzz controller, a controller that is available with the BVM implementation and can be used with ARGoS3 simulator (Pinciroli et al., 2012), 2. BzzKh4,[4] a controller for KheperaIV[5] robots and 3. ROSBuzz,[6] a controller that can be used with ROS compatible robots. Buzz controllers can be considered more than a HAL wrapper to BVM because some controllers leverage the extensible nature of the language (using custom C/C++ function-based primitives) to provide additional features. For instance, ROSBuzz provides features to Geo-Fence robots (limit the operational space for robots), compute veronoi tessellation for robot groups (a method used to partition the space into sub-groups), exploration primitives (methods to plan an exploration path in unknown spaces), etc.

Programming primitives

The programming primitives are pre-built software packages and constructs that can be used to create a more sophisticated control software for the robots. As mentioned earlier, buzz offers constructs for both bottom-up (programming operations performed on individual robots) and top-down (programming operations performed with groups of robots) programming. Robot-wise operations available in Buzz are:

[4] https://github.com/MISTLab/BuzzKH4.

[5] http://www.k-team.com/khepera-iv.

[6] https://github.com/MISTLab/ROSBuzz.

assignment of variables, loops, branching and function definitions. The use of robot-wise operations is analogous to other scripting languages (like Python). As for the top-down programming primitives Buzz offer: Neighbor management, swarm management and virtual Stigmergy. Each of this programming primitive takes inspiration from natural swarm and virtually replicates a phenomenon from natural swarm intelligence. The basic data types available in Buzz are: nil, Int, float, string, table, closure, swarm and virtual stigmergy. Data types nil, Int, float and string are analogous to other scripting languages. Whereas, tables are the only structured datatype available in Buzz that can be either used as tables or dictionaries. Closures correspond to function pointers that can be stored as global variables and referenced at the execution time. Swarm and virtual stigmergy are primitives for top-down programming, and we will discuss them in the following.

Neighbor management
The neighbor management in Buzz is used either for performing operations with the positional information or communicating information within the robots' neighborhood. Figure 11.9 shows a comparison of a behavior observed in nature (flocking) and artificial behavior (boids rule) performing a similar behavior. Neighbors construct simplifies this implementation by using the function *neighbours.foreach* that loops through all the neighbors of a robot and apply a function. The function applied for each neighbor could compute vectors for all three components (separation, cohesion and alignment) for this neighbor. The result of this operation would be one aggregated vector for each component (separation, cohesion and alignment) that can be averaged to obtain the common heading of the robot. There are also other functions in Buzz that could be leveraged in a neighbor based operation: map, reduce and filter. Communication functions like *neighbours.broadcast* and *neighbours.listen* can be used to broadcast messages in the robots neighborhood. For instance, *neighbours.broadcast* can be used to broadcast the value of a Buzz datatype under a topic and *neighbours.listen* can be used to register a callback function to execute when a message is received from a topic.

11.4.2.1 Swarm Management

Programming heterogeneous swarms are a challenging task, locomotion and sensing used by the different types of robot can be fundamentally different. Consider flying, legged robots and rolling robots, each of these robot types need different kinds of sensors and actuators to realize locomotion. With different kinds of sensors and actuators comes different types of programming constructs to operate the robots. Buzz offers swarm construct that can take into account heterogeneity while programming the robots. Figure 11.10 illustrates swarm construct with a heterogeneous swarm containing flying and ground robots. Sub-groups within the robots can be created to perform robot specific operations; these virtually tagged robot groups are called a swarm. From a programming perspective, *swarm.create()* function can be used to create a virtual group (swarm) and functions like *swarm.select()* and *swarm.join()*

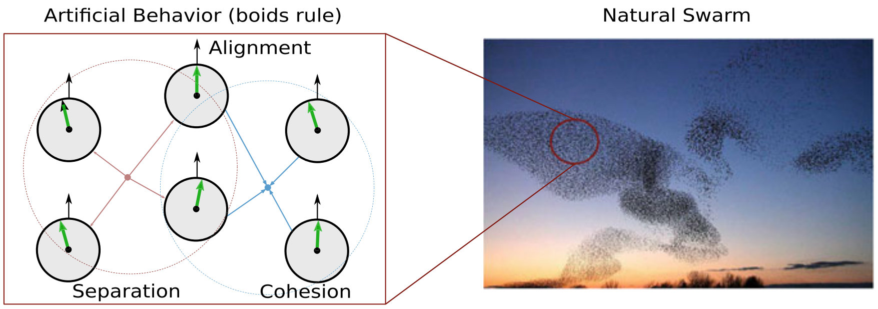


Fig. 11.9 Illustration of flocking: on the right, a starling swarm flocking and on the left, artificial swarm intelligence performing the equivalent behavior using boids rule (separation, alignment and cohesion). This behavior require looping through a robot's neighbors to compute the current movement, neighbors construct in Buzz provides *neighbors.foreach* function to loop through all neighbors and compute the current heading vectors. *Credits* Starling swarm—wikimedia.org/Walter Baxter

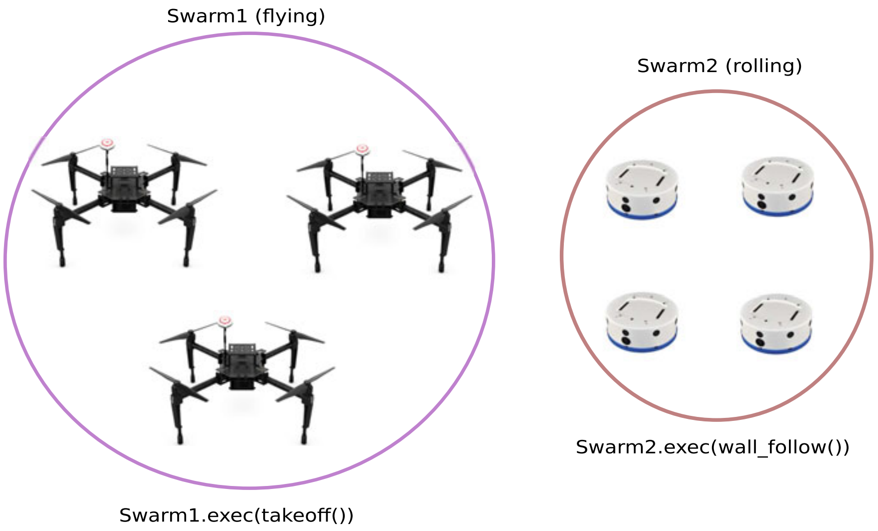


Fig. 11.10 Illustration of swarm construct in Buzz, a group of robots can be virtually tagged to assign group specific behavior, flying robots are assigned takeoff task and ground robots are assigned a wall following behavior

can be applied to join a swarm. As in Fig. 11.10, the function *swarm.exec()* can be used to assign a group specific function to execute for a given swarm.

Virtual stigmergy

Virtual stigmergy is a programming construct derived from natural swarm intelligence called stigmergy. Stigmergy is widely found in insect swarms, consider termites, they change the structure of the mold they build to communicate with other termites, in this case the information flow is environment mediated. Another exam-

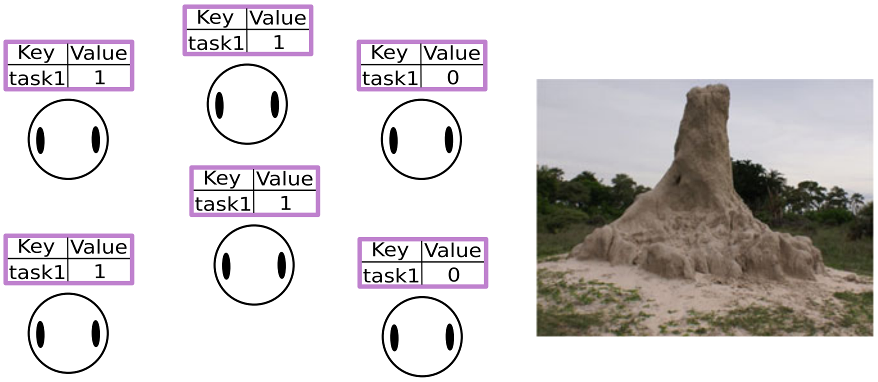


Fig. 11.11 Stigmergy in termites modifying the mold structure to communicate with other termites (right), a virtual implementation of this phenomenon (virtual stigmergy) provide (key, value) tuples to propagate information (left). *Credits* Termite mound—flickr.com/Justin Hall

ple to stigmergy can be found in ants; they spray pheromones to communicate the shortest path to a food source. The environment acts as a medium to relay information to other insects. Virtual stigmergy is a programming construct that allows programmers to replicate this phenomenon in a virtual manner on the robots. Unlike stigmergy in insects, the robots using virtual stigmergy make use of data structures to store and propagate information. From a usage point of view, virtual stigmergy is comparable to shared memory and distributed ledgers, acting as a black board for writing information from one robot and reading it on other robots. Figure 11.11 illustrates virtual stigmergy on robots by making comparison to termite swarms. In programming robots, virtual stigmergy table can be created using *stigmergy.create()*, *stigmergy.put()* can be used to add or modify entries and *stigmergy.get()* can be used to read the latest value. The internal implementation of virtual stigmergy optimizes the information to be broadcast to achieve a guaranteed network wide propagation. The information flow happens in a gossip-based fashion from one robots' neighborhood to another until a unified information is present in the whole group.

11.5 Deployment of Real-World Swarm Systems

Swarm robotics is a very young field of robotics that has received an increasing attention over the past decade due to its inherent benefits. However, the field has not matured enough to have robust real-world deployments, mainly due to the fact that some of the underlying engineering concepts are not completely clear for full autonomy. These challenges have given rise to creation of technologies that can allow humans to supervise and manage the system once deployed.

11.5.1 Human Swarm Interaction

Rapid advances in artificial intelligence are driving the adoption of robotics and automation in transport and logistics, providing new solutions to highway systems (Shladover, 2018), passenger transport (Pavone, 2015), last-mile delivery (Grippa et al., 2019), and automated warehouses (Enright & Wurman, 2011). For the foreseeable future, humans will remain indispensable to supervise and manage such fleets because we are transitioning from systems that are generally already in use; technology gaps prevent us from performing all of the required functions autonomously; and particularly in visible, safety-critical applications, society's trust in decentralized technology will be earned gradually. However, integrating increasingly sophisticated AI techniques leads to increasingly opaque robot control programs. Furthermore, human supervisors' cognitive capacities are challenged (and eventually exceeded) as the size of autonomous fleets grows. The difficulty of ensuring operational performance is compounded when incoming information is scattered, delayed, asynchronous or unreliable. These factors lead to increased pressure on human supervisors' cognitive resources and their ability to maintain situational awareness, detect problems and make successful decisions. There are some methodologies and approaches (St-Onge et al., 2019b) for the supervision of AI-driven swarm systems, deployed across domains such as transportation and logistics.

Given that the operator is indispensable in a robotic fleet to solve complex tasks and communicate with the swarm, the major focus in the field of human swarm interaction (HSI) are the following: 1. Operator cognitive complexity, 2. Communication with the swarm, 3. Control architectures, 4. Level of autonomy and 5. Methods to interact with operator. All of these modules that the field focus on are tightly coupled with one another, for instance, the level of communication of robot states depends on the level of autonomy, which in turn affects the operator effort to control the swarm. A detailed consideration to the concepts in HSI can be found in Kolling et al. (2015).

Operator cognitive complexity
In the field of computer science, the term computational complexity is defined as the resources (such as time and memory) required to solve an algorithmic problem. The required resources are generally considered to be a function of the size of the input. Computational complexity is used to classify the solvable computational algorithm from the unsolvable ones. Higher computational complexity algorithm might work reasonably for smaller number of inputs and fail for larger number of inputs. In HSI, a similar concept exists called the cognitive complexity for the robot control task; instead of the algorithm, an operator is replaced. The main task of an operator in a swarm system is to supervise and manage the robots by performing a sequence of actions on observation of a robot status. Operators cognitive load can be defined as the complexity of actions to perform by the operator on observation of a status. The analogy between computational complexity and cognitive complexity was first drawn in Lewis (2013).

Consider a group of aerial robots are performing a search operation in the forest to locate human survivors, when the robots are managed individually by the operator (checking each of the robot camera feed individually for a human and sending commands to further explore) then the cognitive complexity of this mission is O(n). Conversely, when the operator deploys the robots and selects a search area, the robots subdivide the tasks autonomously, run a human detection algorithm internally using the camera feeds and send the operator of only a possible human detection for verification then the complexity here is O(1), which is the minimal possible cognitive complexity. Another term that relates to cognitive complexity is the negligence tolerance, the time required by the robots to show performance degradation when left unattended. For optimal operation of the fleet, the operator has to attend to the robots before negligence tolerance time. In reality, the cognitive complexity of the system lie between O(1) and O(n) could also be sometimes worse than O(n), when the operator has to deal with a cascade of tasks for a given robot in the swarm.

Communication with the swarm
An operator communicating with the swarm is an essential routine in real-world missions, the current level of robot autonomy demands an operator to be present in the system. An operator generally use a specialized device called the base station to communicate with the swarm. There are two types of communication that might be necessary between a base station and the operator: 1. The operator has to relay high-level goals to the system (commands) and 2. Operator has to obtain situational awareness on the robot fleet (states). Realizing both the goals require a reliable communication infrastructure within the system. Maintaining a reliable communication among the robots in the fleet is a challenging problem. The robots need to move to perform their mission, the movement in turn results in the change of communication topology, the swarm needs to realize the communication topology change for information propagation. One common approach to communication in robot swarms is to design a connectivity maintenance algorithm that will maintain a desired level of connectivity in the swarm allowing a base station to connect to the swarm.

Control architecture
The control architecture used in the swarm system defines the possible controls the operator can have over the system. Control architecture used in the system might influence the operator cognitive complexity, since it limits the possible controls the operator can have over the system. The desired cognitive complexity of controlling a swarm system is O(1), where the operator treats the swarm as a whole, as if it were a single robot with complex dynamics. The current types of control architectures available require more fine grained interactions than swarm-level interaction, demand the operator to interact with sub-groups of robots. Some of the common control architectures used in robot swarms are:

1. Behavior library: Where a set of behaviors are implemented for the robots and the operator selects an appropriate behavior from the behavioral set based on the current situational awareness.
2. Parameter adaptation: A generic behavior is implemented initially and the operator is left to adapt the parameters of the system to control the robotic swarm.

3. Environment mediated control: The operator is made a part of the swarm and interacts with the swarm through the environmental medium (with modalities like gesture control).
4. Leader based control: A selective set of robots are assigned a leader role and the rest of the robots follow the leader robots, the operator continuously interacts with leader robot to control the swarm system. For example: the operator could teleop the leader robots to control the swarm.

Level of autonomy
The level of autonomy (LOA) of a swarm system can be defined as the degree to which the swarm system can make decisions on its own without external support (like an operator). LOA is generally defined through a 10 point scale, initially proposed by Sheridan and Verplank (1978). A scale of 1 defines the swarm to take absolutely no decisions and actions; the operator must perform all the tasks in the system. Conversely, a scale of 10 denotes the system completely disregards the human and performs all actions exclusively autonomously. It is commonly referred (Kolling et al., 2015) that the swarm system lie somewhere in or above a scale of 7, which means the system performs actions autonomously and informs the humans of the choices. The level of autonomy has no influence on the amount of situational awareness an operator acquires to interact with the swarm.

Methods to interact with operator
The method of interaction with the operator is an important factor to consider in system with operators and highly influence the cognitive complexity of the system. There are two methods to operator swarm interaction: 1. Remote interaction and 2. Proximal interactions. In remote interactions, the operator is considered to be monitoring a remote control node called the ground station. The ground station is a specialized computer that is used to obtain situational awareness on the robots mission and send commands back to the system to provide them with directives. Proximal control is another paradigm that considers the operator to be a physical part of the swarm as a special swarm member and these specialized swarm members provide directives to the swarm. Some approaches to proximal control are using gesture control, voice control and expressive motion. In these approaches, the user performs a certain gesture or voice command, which in turn creates a local interaction with the swarm to perform a task. However, the level of control that can be achieved with proximal control is minimal, since the swarm is controlled as a whole.

Human swarm interaction in Buzz
Within the framework of Buzz, several HSI approaches (St-Onge et al., 2019a, 2019b) have been designed to facilitate the operator interaction with the swarm. These works generally use ROSBuzz executing a Buzz script to realize the operator interactions. The operator is considered to be a virtual swarm member and the system uses a ground station to communicate with the operator (remote interaction). The ground station being a virtual swarm member, deploy a similar buzz script and receiving the same states that every other robot in the swarm is receiving. This system has been tested with two types of ground stations: 1. A traditional computer node and

2. A tangible robot fleet interface. A traditional computer node in this setup use a specialized visualization software to visualize and command the swarm. In tangible robot fleet interface, the operator uses a table top map with miniature robots indicating the status of the robots in the swarm. These miniature robots are used to interact with the swarm deployed in the field. The idea behind the use of tangible interface is that, the operator modifies a replica of the swarm, which in turn applies the changes to the actual swarm. The infrastructure in St-Onge et al. (2019a) was used within the framework of Pangaea-X in Lanssorate Spain, where a group of astronauts used the above elaborated interface (computer node and tangible interface) to control the swarm, while the cognitive load on the astronauts was evaluated.

11.5.2 Data Management, Communication and Mobility

In general, multi-robot systems need to collect large amounts of data from their environment, and often these data need to be aggregated, shared and distributed. Consider the task of distributed map merging (Mangelson et al., 2018) and inter-robot loop-closure detection in simultaneous localization and mapping (SLAM) (Lajoie et al., 2020), where robots need to exchange large amounts of data in the form of map fragments and/or pose graphs along with certain key-frame images. Many multi-robot systems are designed to share state information and commands, but their communication infrastructure is often too limited for significant data transfers. A mechanism called SOUL (Varadharajan et al., 2020a) allows members of a fully distributed system to share data with their peers. SOUL leverage a BitTorrent-like strategy to share data in smaller chunks, or datagrams, with policies that minimize reconstruction time. The main challenges addressed in this approach are: 1—cope with dynamic network topologies, 2—optimize the data fragmentation and reconstruction, and 3—optimize the distribution of the datagrams (chunks of injected data). Since peer-to-peer (P2P) file sharing mechanisms are well established in literature, with ample research to demonstrate their robustness and scalability (Reid, 2015), this method leverages some of their strategies (e.g., with the use of distributed hash tables) and integrates additional concepts from decentralized robotic systems. There are few other methods like Swarm mesh (Majcherczyk & Pinciroli, 2020) that provide location based data storage, referred to as spatial consensus to allow robot in a swarm to leverage the storage space on all robots.

The key principle that needs to be addressed for real-world deployments is addressing the perception-action-communication loop in robot swarm. Real-world robot deployments need to perform the following cascading action loop: to perceive the environment, estimate its state, perform an action, communicate its state to its neighbors. This cascading sequence of actions affects the other robots in the swarm and hence is a tightly coupled state that affects each other. A control software designer must consider the presence of perception-action-communication loop at design time.

The ability of a swarm to coordinate and exchange information depends largely on the underlying communication graph. A reliable communication infrastructure allows the robots to exchange information at any time. However, real deployments include many potential sources of failures (environmental factors, mobility, wear and tear, etc.) that can break connectivity and compromise the mission. The underlying assumption taken by several works (St-Onge et al., 2017) includes the robots ability to exchange information. There are two general approaches to connectivity maintenance in multi-robot systems: strict end-to-end connectivity (Stephan et al., 2017) or relaxed intermittent connectivity (Kantaros et al., 2019). Many of these approaches are either computationally intensive or cannot integrate the presence of an operator. There are some alternatives that use lightweight algorithm (Varadharajan et al., 2020b) allowing a heterogeneous group of robots to navigate to a target in complex 3D environments while maintaining connectivity with a ground station by building a chain of robots. The fully decentralized algorithm is robust to robot failures, can heal broken communication links and exploits heterogeneous swarms: when a target is unreachable by ground robots, the chain is extended with flying robots.

11.5.3 Fault Handling

When multi-robot systems are deployed in real-world scenarios, there is an increasing concern regarding the safety and reliability of the system. Robots that are faulty could potential harm humans or infrastructure. The robot control designed for the robots needs to explicitly design mechanisms that can tolerate some common malfunctions at the minimum. Faults in robotic hardware are inevitable, reliable mechanism incorporation within the control software could minimize the risks caused by faulty hardware. There are generally two kinds of robot failures: 1. Endogenous and 2. Exogenous faults. Endogenous faults are generally faults that occurs within the robotic hardware and exogenous faults are the faults that occur as a result of factors in the environment, and the robots interaction with the environment.

There are two kinds of approaches to detect faults in robot swarms: 1. Introspection and 2. Extrospection. In introspection, the robots run some kind of internal diagnostics to determine if the hardware is faulty. Extrospection is using the diagnostics of the neighboring robots to determine if a given robot is faulty. Some kinds of faults can be addressed using introspection and others require extrospection, currently resolved using an operator in the loop. Extrospection is an interesting solution, when dealing with multi-robot systems, for instance, a deviation from normal operation of a robot can be detected by other robots. Some existing approaches to fault detection are:

- Communication based (Christensen et al., 2009; Ozkan et al., 2010): Robots inability to communicate is detected by using periodic ping messages between the robots. These method can only detect completely failed robots or robots with communication issues.
- Model based (Millard et al., 2014): Robots use a model to compare their behavior to determine normal operation.

- Task effort based (Lau et al., 2011): Robots compute their contribution to the fulfilment of the task and estimate if they are contributing to the global task to determine their fault state.
- Online methods (Tarapore et al., 2017): An online classification model is learnt to distinguish between faulty and normal behavior of the robots; robots evaluate their behavior with the neighbor to determine faultiness. There are some methods that use a immunology inspired models to predict faulty robots (Tarapore et al., 2015).

11.6 Chapter Summary

This chapter provides an introduction to the different types of multi-robot systems and introduces the task allocation problem used in assigning tasks to different robots in a multi-robot system. A particular concentration is given to decentralized systems and various methods available to design decentralized control software. Fundamentals of programming a robotic swarm is discussed using the Buzz programming language. Toward the end of the chapter, a discussion is made regarding the necessity of an operator in a swarm system and the challenges toward the real-world deployment of swarm systems.

11.7 Chapter Revision

Question #1
What are the fundamental differences between centralized, distributed and decentralized systems?

Question #2
What are the design rules followed in the design of decentralized system?

Question #3
What are different methods to design control software for robot swarms?

Question #4
What are the desirable properties expected from a swarm programming language?

Question #5
What is the reference communication and execution model used in Buzz?

Question #6
When a operator needs to send individual commands to each robot in a swarm, what is the operator cognitive complexity in this situation?

Question #7
What are the types of information that needs to be exchanged between an operator and a robot swarm?

Question #8
How many point scales are generally used to identify the level of autonomy in a swarm robotic system?

Question #9
What are the common methods used for operator interaction with the swarm?

Question #10
Why is it important to maintain a desired communication topology in robot swarms?

Question #11
What are the types of faults that arise in robot swarms?

11.8 Further Reading

For further information on distributed multi-robot methods such as auction and voting, we refer the reader to Chaps. 9 and 23 of Easley et al. (2012), respectively. For more information on decentralized control software design, we refer the reader to Francesca and Birattari (2016). For more information on the Buzz programming, we refer the reader to Beltrame (2016). As a further reading on human swarm integration, we refer the reader further reading on human swarm integration, we refer the reader to Kolling et al. (2015).

References

Beal, J., & Bachrach, J. (2006). Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems, 21*(2), 10–19.

Bell, J. E., & McMullen, P. R. (2004). Ant colony optimization techniques for the vehicle routing problem. *Advanced Engineering Informatics, 18*(1), 41–48.

Bonabeau, E., Theraulaz, G., & Dorigo, M. (1999). *Swarm intelligence*. Springer.

Chang, L., Liao, C., Lin, W., Chen, L. L., & Zheng, X. (2012). A hybrid method based on differential evolution and continuous ant colony optimization and its application on wideband antenna design. *Progress in Electromagnetics Research, 122*, 105–118.

Christensen, A. L., OGrady, R., & Dorigo, M. (2009). From fireflies to fault-tolerant swarms of robots. *IEEE Transactions on Evolutionary Computation, 13*(4), 754–766.

Dantu, K., Kate, B., Waterman, B., Bailis, P., & Welsh, M. (2011). Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems* (pp. 121–134).

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine, 1*(4), 28–39. https://doi.org/10.1109/MCI.2006.329691

Dorigo, M., Theraulaz, G., & Trianni, V. (2021). Swarm robotics: Past, present, and future. *Proceedings of the IEEE, 109*(7), 1152–1165.

Easley, D., & Kleinberg, J. (2012). *Networks, crowds, and markets*. Cambridge Books.

Enright, J. J., & Wurman, P. R. (2011). Optimization and coordinated autonomy in mobile fulfillment systems. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*. Citeseer.

Fan, T., Long, P., Liu, W., & Pan, J. (2020). Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios. *The International Journal of Robotics Research, 39*(7), 856–892.

Francesca, G., & Birattari, M. (2016). Automatic design of robot swarms: Achievements and challenges. *Frontiers in Robotics and AI, 3*, 29.

Gomes, J., Urbano, P., & Christensen, A. L. (2013). Evolution of swarm robotics systems with novelty search. *Swarm Intelligence, 7*(2), 115–144.

Grippa, P., Behrens, D. A., Wall, F., & Bettstetter, C. (2019). Drone delivery systems: Job assignment and dimensioning. *Autonomous Robots, 43*(2), 261–274.

Hecker, J. P., Letendre, K., Stolleis, K., Washington, D., & Moses, M. E. (2012). Formica ex machina: Ant swarm foraging from physical to virtual and back again. In *International Conference on Swarm Intelligence* (pp. 252–259). Springer.

Hwang, K., Dongarra, J., & Fox, G. C. (2013). *Distributed and cloud computing: From parallel processing to the internet of things*. Morgan Kaufmann.

Iocchi, L., Nardi, D., Piaggio, M., & Sgorbissa, A. (2003). Distributed coordination in heterogeneous multi-robot systems. *Autonomous Robots, 15*(2), 155–168.

Kacprzyk, J., Merigó, J. M., Nurmi, H., & Zadrozny, S. (2020). Multi-agent systems and voting: How similar are voting procedures. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems* (pp. 172–184). Springer.

Kantaros, Y., Guo, M., & Zavlanos, M. M. (2019). Temporal logic task planning and intermittent connectivity control of mobile robot networks. *IEEE Transactions on Automatic Control, 64*(10), 4105–4120. https://doi.org/10.1109/tac.2019.2893161

Karpov, V., Migalev, A., Moscowsky, A., Rovbo, M., & Vorobiev, V. (2016). Multi-robot exploration and mapping based on the subdefinite models. In *International Conference on Interactive Collaborative Robotics* (pp. 143–152). Springer.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of ICNN'95—International Conference on Neural Networks* (Vol. 4, pp. 1942–1948). https://doi.org/10.1109/ICNN.1995.488968

Kolling, A., Walker, P., Chakraborty, N., Sycara, K., & Lewis, M. (2015). Human interaction with robot swarms: A survey. *IEEE Transactions on Human-Machine Systems, 46*(1), 9–26.

Kuckling, J., Ligot, A., Bozhinoski, D., & Birattari, M. (2018). Behavior trees as a control architecture in the automatic modular design of robot swarms. In *International Conference on Swarm Intelligence* (pp. 30–43). Springer.

Lajoie, P. Y., Ramtoula, B., Chang, Y., Carlone, L., & Beltrame, G. (2020). Door-slam: Distributed, online, and outlier resilient slam for robotic teams. *IEEE Robotics and Automation Letters, 5*(2), 1656–1663.

Lau, H., Bate, I., Cairns, P., & Timmis, J. (2011). Adaptive data-driven error detection in swarm robotics with statistical classifiers. *Robotics and Autonomous Systems, 59*(12), 1021–1035.

Lewis, M. (2013). Human interaction with multiple remote robots. *Reviews of Human Factors and Ergonomics, 9*(1), 131–174.

Luna, R., & Bekris, K. E. (2011). Efficient and complete centralized multi-robot path planning. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3268–3275). IEEE.

Majcherczyk, N., & Pinciroli, C. (2020). SwarmMesh: A distributed data structure for cooperative multi-robot applications. In *2020 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 4059–4065). IEEE.

Mangelson, J. G., Dominic, D., Eustice, R. M., & Vasudevan, R. (2018). Pairwise consistent measurement set maximization for robust multi-robot map merging. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 2916–2923). https://doi.org/10.1109/ICRA.2018.8460217

McLurkin, J. (2009). Experiment design for large multi-robot systems. In *Robotics: Science and Systems, Workshop on Good Experimental Methodology in Robotics*, Seattle, WA.

Michael, N., Zavlanos, M. M., Kumar, V., & Pappas, G. J. (2008). Distributed multi-robot task assignment and formation control. In: *2008 IEEE International Conference on Robotics and Automation* (pp. 128–133). IEEE.

Millard, A. G., Timmis, J., & Winfield, A. F. (2014). Run-time detection of faults in autonomous mobile robots based on the comparison of simulated and real robot behaviour. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3720–3725). IEEE.

Mottola, L., Moretta, M., Whitehouse, K., & Ghezzi, C. (2014). Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems* (pp. 177–190).

Nolfi, S., & Floreano, D. (2000). *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT Press.

Otte, M., Kuhlman, M. J., & Sofge, D. (2020). Auctions for multi-robot task allocation in communication limited environments. *Autonomous Robots, 44*(3), 547–584.

Ozkan, M., Kirlik, G., Parlaktuna, O., Yufka, A., & Yazici, A. (2010). A multi-robot control architecture for fault-tolerant sensor-based coverage. *International Journal of Advanced Robotic Systems, 7*(1), 4.

O'Grady, R., Christensen, A. L., & Dorigo, M. (2012). SWARMORPH: Morphogenesis with self-assembling robots. In *Morphogenetic engineering* (pp. 27–60). Springer.

Pavone, M. (2015). Autonomous mobility-on-demand systems for future urban mobility. In *Autonomes Fahren* (pp. 399–416). Springer.

Petersen, K. H., Nagpal, R., & Werfel, J. K. (2011) TERMES: An autonomous robotic system for three-dimensional collective construction. *Robotics: Science and Systems, VII*.

Pianini, D., Viroli, M., & Beal, J. (2015). Protelis: Practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (pp. 1846–1853).

Pinciroli, C., & Beltrame, G. (2016). Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 3794–3800). IEEE.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Birattari, M., Gambardella, L. M., & Dorigo, M. (2012). ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence, 6*(4), 271–295.

Pinedo, M. (2012). *Scheduling* (Vol. 29). Springer.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., & Ng, A. Y. (2009). ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, Kobe, Japan (Vol. 3, p. 5).

Reid, N. (2015). *Literature review: Purely decentralized P2P file sharing systems and usability* (Technical report). Rhodes University, Grahamstown.

Schwager, M., Dames, P., Rus, D., & Kumar, V. (2017). A multi-robot control policy for information gathering in the presence of unknown hazards. In *Robotics research* (pp. 455–472). Springer.

Sheng, W., Yang, Q., Tan, J., & Xi, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems, 54*(12), 945–955.

Sheridan, T. B., & Verplank, W. L. (1978). *Human and computer control of undersea teleoperators* (Technical report). Massachusetts Institute of Technology Cambridge Man-Machine Systems Lab.

Shladover, S. E. (2018). Connected and automated vehicle systems: Introduction and overview. *Journal of Intelligent Transportation Systems, 22*(3), 190–200.

St-Onge, D., Kaufmann, M., Panerati, J., Ramtoula, B., Cao, Y., Coey, E. B., & Beltrame, G. (2019a). Planetary exploration with robot teams: Implementing higher autonomy with swarm intelligence. *IEEE Robotics & Automation Magazine, 27*(2), 159–168.

St-Onge, D., Varadharajan, V. S., & Beltrame, G. (2019b). Tangible robotic fleet control. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems* (pp. 2387–2389).

Stephan, J., Fink, J., Kumar, V., & Ribeiro, A. (2017). Concurrent control of mobility and communication in multirobot systems. *IEEE Transactions on Robotics, 33*(5), 1248–1254. https://doi.org/10.1109/TRO.2017.2705119

Styø, K. (2001). Using situated communication in distributed autonomous mobile robotics. *SCAI, Citeseer, 1*, 44–52.

Tarapore, D., Lima, P. U., Carneiro, J., & Christensen, A. L. (2015). To err is robotic, to tolerate immunological: Fault detection in multirobot systems. *Bioinspiration & Biomimetics, 10*(1), 016014.

Tarapore, D., Christensen, A. L., & Timmis, J. (2017). Generic, scalable and decentralized fault detection for robot swarms. *PLoS ONE, 12*(8), e0182058.

Varadharajan, V. S., St-Onge, D., Adams, B., & Beltrame, G. (2020a). Soul: Data sharing for robot swarms. *Autonomous Robots, 44*(3), 377–394.

Varadharajan, V. S., St-Onge, D., Adams, B., & Beltrame, G. (2020b). Swarm relays: Distributed self-healing ground-and-air connectivity chains. *IEEE Robotics and Automation Letters, 5*(4), 5347–5354. https://doi.org/10.1109/LRA.2020.3006793

Wurm, K. M., Stachniss, C., & Burgard, W. (2008). Coordinated multi-robot exploration using a segmentation of the environment. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1160–1165). IEEE.

Xing, L. N., Chen, Y. W., Wang, P., Zhao, Q. S., & Xiong, J. (2010). A knowledge-based ant colony optimization for exible job shop scheduling problems. *Applied Soft Computing, 10*(3), 888–896.

Yan, Z., Jouandeau, N., & Cherif, A. A. (2010). Sampling-based multi-robot exploration. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, VDE (pp. 1–6).

Vivek Shankar Varadharajan is a Ph.D. candidate in the Department of Computer Engineering and Software Engineering at École Polytechnique de Montréa. Varadharajan obtained his M.Sc. degree in Automation and Robotics from Technical University of Dortmund in 2015. He is a full-stack robotics developer and has vast experience in equipping robotic platforms with SLAM algorithms, navigation/traversability algorithms and robotic behaviors. His research interests include distributed robotics, multi-robot systems, machine learning, artificial intelligence and Cyber-Physical systems. During his study, he has won several prizes at technical contents, hackathon, poster presentation and demonstrations. He was a team member of CoStar that took part in the DARPA subterranean challenge along with members from NASA Jet Propulsion Laboratory. He has supervised over 5 interns during his Ph.D. He is a recipient of a BSFD student scholarship from École Polytechnique de Montréal.

Giovanni Beltrame is a full time professor in the Department of Computer Engineering and Software Engineering at École Polytechnique de Montréal. Beltrame obtained his Ph.D. in Computer Engineering from Politecnico di Milano, in 2006 after which he worked as microelectronics engineer at the European Space Agency on a number of projects spanning from radiation-tolerant systems to computer-aided design. In 2010 he moved to Montreal, Canada where he is currently Professor at Polytechnique Montreal with the Computer and Software Engineering Department. He was also Visiting Professor at the University of Tübingen in 2017/2018. Dr. Beltrame directs the MIST Lab, with more than 20 students and postdocs under his supervision. He has completed several projects in collaboration with industry and government agencies in the area of robotics, disaster response, and space exploration. His research interests include modeling and design of embedded systems, artificial intelligence, and robotics, on which he has published research in top journals and conferences.