ECOLE
POLYTECHNIQUE
DE BRUXELLES

Optimization in the automatic modular design of control software for robot swarms

Thesis presented by Jonas KUCKLING
in fulfilment of the requirements of the PhD Degree in Engineering and Technology (”Docteur en Sciences de l’Ingénieur et Technologie”)
Année académique 2022-2023

Supervisor: Professor Mauro BIRATTARI
Co-supervisor: Professor Thomas STÜTZLE
IRIDIA—Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Articifielle

Thesis jury :

Marco DORIGO (Université libre de Bruxelles, Chair)
Heiko HAMANN (Universität Konstanz)
Leslie PÉREZ CÁCERES (Pontificia Universidad Católica de Valparaíso)

European Commission
Horizon 2020
European Union funding
for Research & Innovation
LA LIBERTÉ DE CHERCHER
European Research Council
Established by the European Commission

The thesis

Exploring alternative optimization algorithms allows us to understand the influence they have on the automatic modular design of control software for robot swarms.

Summary

The aim of this dissertation is to investigate the role of optimization in the automatic modular design of control software for robot swarms. One of the main challenges in swarm robotics is to design the behavior of the individual robots so that a desired collective mission can be performed. Optimization-based design methods utilize an optimization algorithm to search for well-performing instances of control software. In optimization-based design, past research has mainly focused on proving the feasibility of optimization-based design methods for given missions. With this approach, researchers could tackle a wide range of missions. However, only a few works compare the role of the components of any chosen optimization-based design method. In particular, very little attention has been devoted to the optimization algorithm, arguably the central element in optimization-based design. In the context of my research, I focused on automatic modular design, an optimization-based design approach that combines modules into higher-level control architectures. Automatic modular design has shown to produce control software that not only performs well in simulation but that also transfers well into reality.

In this dissertation, I present a study of different types of optimization algorithms: local-search, model-free racing, and model-based. I defined three automatic modular design methods and compared them against state-of-the-art methods from the literature. I assessed and compared these design methods in experiments for several missions, both in simulation and on real robots. In particular, I showed that, while the choice of the optimization algorithm has an impact on the performance of the generated control software, it appears to not compromise the ability to cross the reality gap satisfactorily.

The work presented in this dissertation represents a first step towards systematically investigating the role of optimization in optimization-based design. More work is still needed to further our understanding of it.

Author's declaration

This dissertation presents an original work that has never been submitted to Université libre de Bruxelles or any other institution for the award of a doctoral degree. Some parts of this dissertation are based on a number of peer-reviewed publications that I, together with my respective co-authors, have published in the scientific literature.

Acknowledgments

I acknowledge support from the following organizations and projects: from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (DEMIURGE Project, grant agreement No 681872); from Belgium's Wallonia-Brussels Federation through the ARC Advanced Project GbO-Guaranteed by Optimization; and from the Belgian Fonds de la Recherche Scientifique–FNRS via the crédit d'équippement SwarmSim. I also acknowledge personal support from the Belgian Fonds de la Recherche Scientifique–FNRS through an ASP fellowship and from Wallonie-Bruxelles International through a WBI.World fellowship.

I would like to thank my two supervisors: Prof. Mauro Birattari and Prof. Thomas Stützle. They have taught me how to become the researcher that I am today. This work would not have been possible without them. Mauro, you have taught me to express my thoughts clearly and the value a single word can hold. Thomas, I admire your patience and kindness. I would also like to acknowledge the support and kind welcoming of Prof. Holger Hoos, with whom I have worked on the research that forms a part of this dissertation.

My thanks also go the all the IRIDIAns. The accumulation of knowledge, different points of views and life experiences was truly enriching. In particular, I would like to thank the DEMIURGE team: Dr. Darko Bozhinoski, Dr. Federico Pagnozzi, Dr. Ken Hasselmann, Dr. Antoine Ligot, Dr. Fernando Mendiburu, Muhammad Salman, David Garzón Ramos, Miquel Kegeleirs, Guillermo Legarda Herranz, and Ilyes Gharbi. It has been my great pleasure to work together with all of you. You have shown me that the whole is greater than the sum of its parts.

I would like to extend my thanks also to those who have accompanied me on this journey outside of work: The always cheerful Christian Camacho Villalón, Guillaume Levasseur, the master of cheese, Peter Kraus, who seems to know and own every board game in existence, and Weixu "Harry" Zhu, the nicest Slytherin I have ever met. Thank you also to Sara and Salman, who have been truly great friends. It was a pleasure that all of you were part on my journey in this stage of life. Last but not least, David Garzón Ramos. David, you have not only been a colleague and flatmate but also a great friend. And even though our discussion

always tended to gravitate towards robotics, they have been invaluable to me. I also thank everyone not explicitly mentioned here, who contributed to the success of this work from near or far.

Finally, I wish to thank my family. Their support on every level has been amazing and without it I would not be here today. Mami, Papi, Hannah, Linus, thank you for everything!

My last thanks go to my dearest Imène. I cannot express enough my gratitude for your support, understanding, and love. I am glad to have had you by my side during all this time and I cannot wait to share with you what will come next.

Jonas

Contents

Chapter 1

Introduction

Mechanical apparatuses and artificial creatures that can act apparently autonomously have fascinated humanity for a long time. For example, Aristotle wrote about "tool[s that] could perform [their] own work when ordered, or by seeing what to do in advance" (Aristotle, 1967) as a means of replacing slavery. Several Jewish myths tell of the golem: an artificial creature, created to perform the will of its creators. Rossum's Universal Robots is a 1920 play centered on a factory creating artificial humans. It was the first work to introduce the term "robot". In modern media, robots have become ubiquitous, e.g., the droids from Star Wars, the Terminator, or Wall-E.

Outside of the media, through the ages, many mechanical constructs have been built that try to emulate intelligent behaviors. Maybe the most well-known automaton pretending to be intelligent is the Mechanical Turk. Built in 1770, it was a machine that seemingly could play chess and even challenged Napoleon I of France and Benjamin Franklin. However, these early automatons were not autonomous. Either they could only perform a pre-designed action and could not act independently based on their perception of the environment, or they were controlled by another entity. In fact, the Mechanical Turk was secretly operated by a chess grandmaster hidden inside.

The first autonomous robots were the "tortoises" of Walter (Walter, 1950). They could perceive light conditions in the environment and use these to control their motion. Since then, robots have found applications in various areas (for example, see Figure 1.1). Robotic manipulators play a vital role in assembly lines in the industry (KUKA Group, 2021), consumer robots like robotic vacuum cleaners have entered households (Pierce, 2018), and drones are being used for photography (The Creative Cloud Team (Adobe), 2017), cinematography (wallonia.be, 2014) and to orchestrate light shows (Cheung, 2017).

To this date, much of the work has been dedicated to the autonomy of single robot systems. Indeed, single robots can perform quite complex feats, such as

Image credit: Intel Corporation.

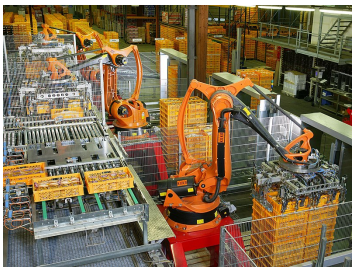


Image credit: KUKA Roboter GmbH.

Figure 1.1: Two examples of real-world applications of robots. *Left:* A group of Intel Shooting Star drones performing at the 2018 Olympic Games in PyeongChang. *Right:* An industrial manipulator handling pallets of bread at a bakery.

performing surgery (Hannaford et al., 2013), performing backflips (Boston Dynamics, 2017; Chignoli et al., 2021), or exploring mars (Witze, 2022). However, the coordination of large robot groups often suffers from the same lack of autonomy as early automatons. For example, in the opening show of the 2020 Olympic Games, Intel used a group of 1824 drones to perform a light show. However, the flight paths of the drones were pre-computed and controlled by an external control station (Cheung, 2017). This is reminiscent of the Mechanical Turk. For the audience, a seemingly autonomous system is displayed, but behind the scenes, it is controlled by another entity.

I believe that in the future, robots will need to interact in large groups and in unknown, unstructured and dynamic environments. Consider the fictitious example of Maximo, an entrepreneur that owns his own robotic moving company. To speed up the moving process and handle heavy or unwieldy items, Maximo brings not only a single robot, but a group of robots. However, the number of robots required for each move is variable and depends among other things on the number of items to move. As the move happens in the apartments and offices of his clients, he cannot rely on external infrastructure. Furthermore, the clients might provide him with a blueprint of the property and estimates of the location of items, but Maximo does not know the exact number and locations of items before he arrives. Additionally, while Maximo takes good care of his robots, it can always happen that a robot ceases to function. Fixing the robot will take some time, but Maximo still wants to finish his contract with the remaining robots. Last, Maximo wants to service as many clients as possible and will distribute his robots between locations. When one location is finished, he will move his robots to the next location, possibly joining other robots that already operate at that location.

Swarm robotics is an approach to control large numbers of robots in an autonomous and decentralized fashion (Şahin, 2005; Beni, 2005; Brambilla et al.,

Image credit: Marco Dorigo et al.

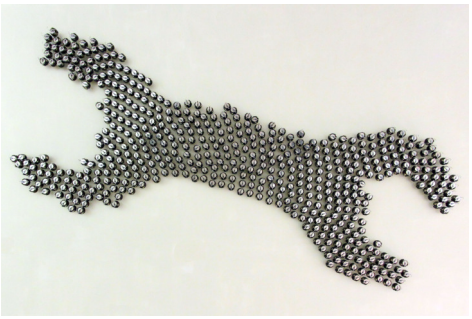


Image credit: Michael Rubenstein.

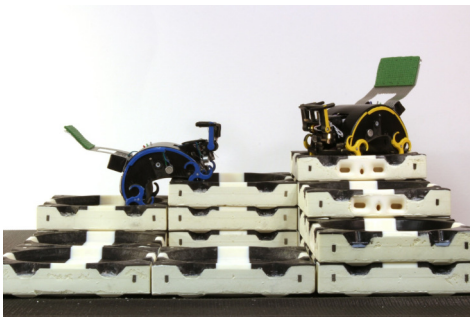


Image credit: Radhika Nagpal.

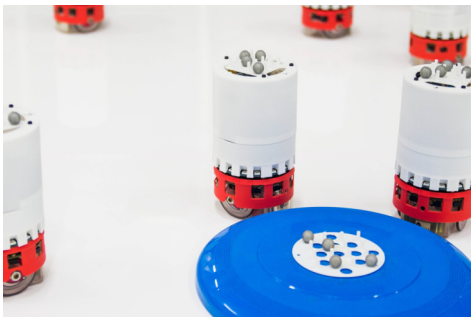


Image credit: Bristol Robotics Lab.

Figure 1.2: Four examples of famous robot swarms. *Top-left:* The Swarmanoid (Dorigo et al., 2013) is a heterogeneous swarm of robots, capable of collaborating to transport items. *Top-right:* Kilobots (Rubenstein et al., 2014) are a low-cost robotic platform and have been used in swarms up to 1024 robots. *Bottom-left:* The TERMES robots (Werfel et al., 2014) are termite inspired and have been used to form a swarm capable of constructing complex structures. *Bottom-right:* The two terraflop swarm (Jones et al., 2018a) is composed by Xpuck robots and provides a large amount of computational power for distributed computational tasks.

2013; Dorigo et al., 2014; Hamann, 2018). A robot swarm is a highly redundant, decentralized and autonomous system composed of relatively simple robots with local sensing and communication capabilities. Due to decentralization, the swarm must be self-organized, that is, the collective behavior cannot be programmed directly, but emerges from the interactions of the individuals in the swarm. For examples of well-known robot swarms, see Figure 1.2.

The principles of swarm robotics lend themselves to the conception and implementation of robotic systems that exhibit several desirable properties (Şahin, 2005; Dorigo et al., 2014; Hamann, 2018). The self-organized nature of robot swarms promotes the design of flexible systems: the swarm can adapt to different and potentially changing environments. Additionally, the redundancy of the swarm

facilitates the creation of systems that are fault tolerant. The failure of any individual robot (or sometimes significant portions of the swarm) will not prevent the swarm from achieving its task. Lastly, as robots only interact with their immediate neighborhood, swarms are often scalable. That is, the addition or removal of robots from the swarm does not significantly affect the performance of the swarm. For a summary of the history, achievements and challenges of the field, see Chapter 2.

Robot swarms offer several advantages, but their decentralized and self-organized nature makes them challenging to design. The requirements for the desired behavior of the swarm are usually expressed at the collective level, yet it is not possible to program the swarm directly. Instead, software needs to be generated for the individual robots. Swarm engineering is the discipline that addresses the design of control software for a swarm of robots (Brambilla et al., 2013). In swarm engineering, the design problem can be phrased as follows: Given a mission-specification—robotic platform, environment and desired collective behavior—create control software, so that the robots perform the desired collective behavior in the environment. Due to the decentralized nature of robot swarms and the lack of global information available to the individual robots, each robot can only act based on the local information that it can perceive. As a result, the designer needs to predict how the local behaviors and local interactions contribute to the emergence of the desired collective behavior. This connection between the local behavior (microscopic level or micro-level) and the collective behavior (macroscopic level or macro-level) is usually non-trivial. Several works are addressing this issue, trying to find a link between the micro-level and the macro-level (see Chapter 2.3 for an overview and discussion of recent approaches). To this date, no general micro-macro link exists, and proposed links are often restricted to specific missions or even specific individual behaviors (Brambilla et al., 2013; Francesca and Birattari, 2016; Schranz et al., 2021).

Based on these theoretical findings, several design formal methods have been proposed (Hamann, 2018; Elamvazhuthi and Berman, 2019). Based on the developed micro-macro links, they allow to derive control software based on the dynamics of the global swarm behavior. Formal methods allow to derive optimal control software for the given mission. However, formal methods require a theoretical understanding of the collective behavior and its dynamics. Developing such a theoretical model is challenging and the success of doing so depends on the skill of the designer. As a result, formal methods exist only for a few missions.

Barring a formal understanding of the swarm dynamics, the most common approach to the design challenge is manual design: a human designer implements the control software for the robot. The designer can refine the control software through a trial-and-error process until they find the result satisfactory. While this design process often yields reasonable results (Francesca and Birattari, 2016), it can be error-prone, costly, time-consuming and the quality of the control software

strongly depends on the expertise of the human designer. Furthermore, there is no guarantee that the performance will reach a satisfactory level within any reasonably available time budget. Recently, several software-engineering techniques and patterns have been proposed, yet they operate usually only in specific missions and restrictive assumptions (Francesca and Birattari, 2016; Bozhinoski and Birattari, 2018; Birattari et al., 2019).

Another alternative design approach is optimization-based design. In optimization-based design, the design problem is transformed into an optimization problem. The optimization problem can be stated as follows: Given a mission specification—robotic platform, mission environment and mission-specific performance measure (objective function)—find an instance of control software that maximizes the objective function. To that end, an optimization algorithm searches the space of all possible instances of control software to find an optimal one.

Optimization-based design can be further classified according to several criteria (see Figure 1.3) (Birattari et al., 2020). One major distinction in optimization-based design is between semi-automatic and fully automatic design. In semi-automatic design, a human designer remains in the loop. That is, the human designer can observe and intervene in the design process, if necessary. For example, the human designer could observe the result found by the optimization algorithm, change some parameters used in the optimization process and restart it with the new parameters. The semi-automatic design terminates when the human designer is satisfied with the generated control software. Semi-automatic design exhibits similar drawbacks as manual design, namely that the quality of the generated control software depends on the human designer and their ability to steer the design process. In fully automatic design, contrarily, the human designer cannot intervene in the design process beyond the mission specification. Therefore, the design process can be considered being *one-shot*.

Another major distinction is between online and offline design. In online design, the design process creates the control software while the swarm performs the mission. This approach has several drawbacks. First, the design process requires access to the robotic hardware and the mission environment. Second, if no prior safety measures have been implemented, the design process might damage the robots when performing dangerous maneuvers. Third, the swarm might not be capable of assessing the performance of its own behavior and might require an external infrastructure that cannot be provided in all circumstances. In offline design, the design process is run in a facsimile of the mission environment. This could be, for example, a mock-up environment, mimicking the mission environment or, more commonly, a simulation of the mission environment. By running the design in a controlled environment instead of the mission environment, the designers can introduce infrastructure to supervise the operation of the swarm during the design process, e.g., a tracking system that tracks the position of the

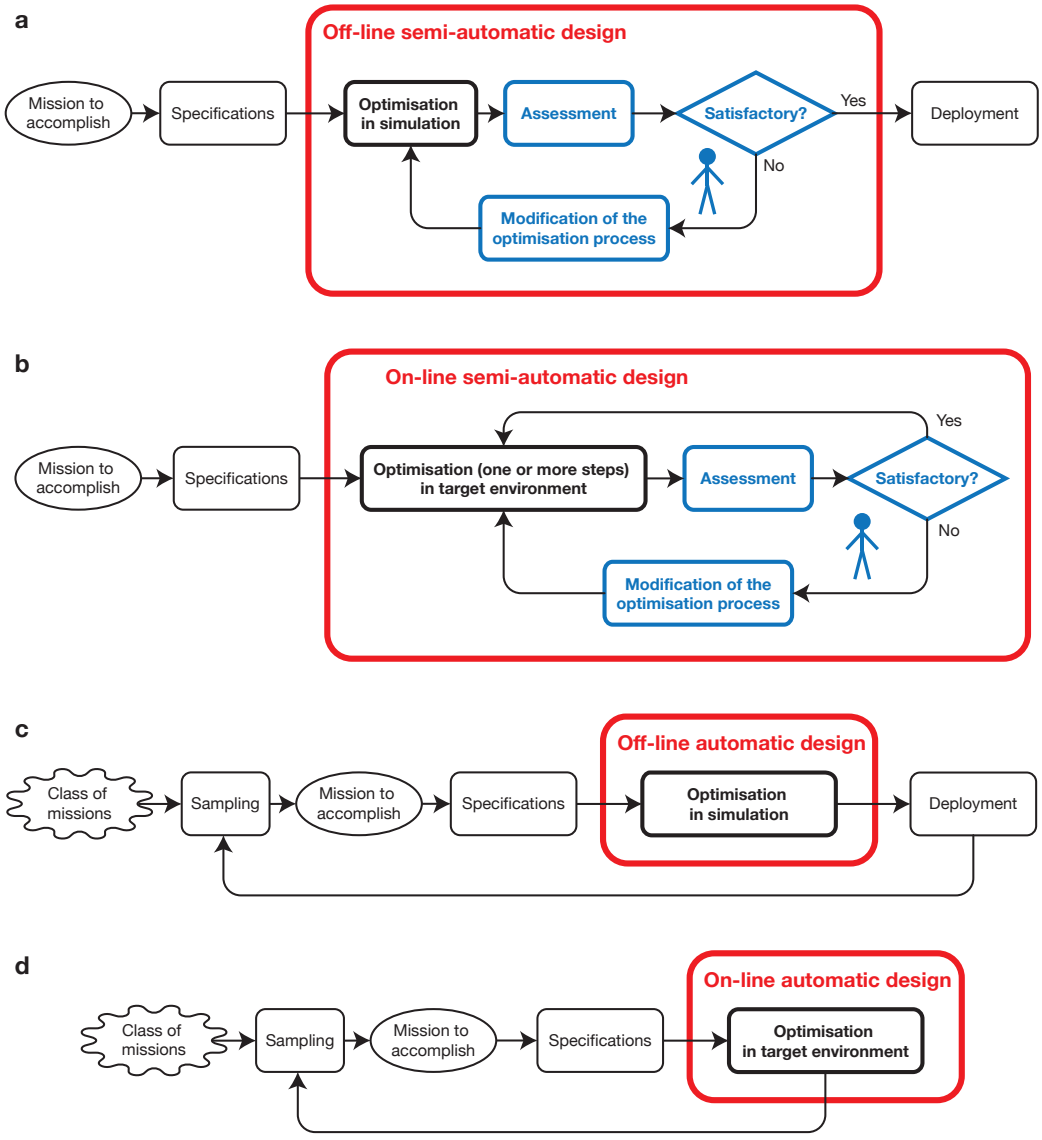


Image originally published by Birattari et al. (2020).

Figure 1.3: Four classes of optimization-based design. In semi-automatic design, a human designer remains in the loop during the design process. In fully automatic design, no human intervention is possible during the design process. Offline design is performed in a stand-in for the target environment, usually a simulation. Online design is performed directly in the target environment.

robots to compute the objective function. Further, by using simulations, the remaining drawbacks of online design can be addressed, as simulations can be run without access to the robotic hardware. Furthermore, simulations can be executed faster than real-time, and they can be parallelized, potentially decreasing the time needed for the design process. However, the use of simulations entails one major drawback: the reality gap.

The reality gap are the inescapable differences between the simulation and reality. It manifests often as a drop in performance when porting software designed in simulation to real robots. It has often been postulated that the reality gap is because of simulations being a too simplistic model of reality. As such, much effort has been devoted to creating more realistic simulators. Recent research, however, suggests that this is not the case and that the effects of the reality gap can be reproduced between two sets of simulators of similar complexities (Ligot and Birattari, 2020, 2022).

As the reality gap does not affect all design methods equally, Francesca et al. (2014) postulated that the reality gap can be seen akin to overfitting in machine learning. The reality gap manifests not because of the simplicity of the simulator, but because of the design process inescapably overfits certain idiosyncrasies of the simulator. They argue that robustness to the reality gap can be seen as similar to the bias-variance trade-off in machine learning. Design methods that can generate control software for potentially every input-output mapping tend to exploit the idiosyncrasies of the simulator to achieve the high specialization. The authors argue that reducing the space of possible input-output mappings should increase the robustness to the reality gap. This can be achieved by restricting the design process to only produce control software that can be obtained by assembling predefined modules.

An automatic modular design method is then composed of several parts: the reference model, the modules, the target architecture and the optimization algorithm. The reference model provides a formalization of the sensors and actuators of the robotic platform. Based on the reference model, the behavioral and conditional modules are defined. The target architecture forms an abstract representation of the possible instances of control software. Through the selection and assembly of modules into the target architecture, it is instantiated to a particular instance of control software. This instantiation will be performed automatically by an optimization algorithm searching for an optimal instance of control software.

Coming back to the previous example of Maximo. He decided to organize his robots into a robot swarm as the properties of robot swarms will benefit his use case. As the robots operate in a location owned by the client, Maximo cannot rely on any infrastructure to control the robots himself. Instead, the robots will need to act autonomously. The self-organization and flexibility of the swarm allows it to adapt to different environments, even if the location is not identical

to how the client described it. The fault tolerance of the system ensures that even if some of Maximo's robots fail, he can still fulfill his contract. The scalability will benefit Maximo's swarm when he adds robots that previously worked on a different location.

As Maximo's clients book his services with little time in advance, Maximo requires a design method that can reliably produce results in a short time. Furthermore, Maximo's clients expect his robots to immediately start performing well, therefore Maximo needs to design the software before deploying his swarm at the location. Consequently, Maximo cannot rely on manual design or online design and would need to use offline design. As it would be not feasible for him to build facsimiles of his clients' locations, Maximo relies on simulations to design the control software for his swarm. However, the reality gap remains a major issue and can cause Maximo's robots to fail the contract. Maximo therefore decided to use an automatic modular design method.

Automatic modular design is a good approach for Maximo for several reasons. First, it is an offline design method that promises to be robust to the reality gap, which is in line with Maximo's needs. Second, as his robots will need to perform similar actions for all clients, Maximo can easily program the modules of the robots. He can test that the modules are robust to the reality gap in small experiments—for example, for example in a small and abstract facsimile environment.

Of the four elements of the design method, the reference model and modules are dictated by the choice of robots and the needs of the clients. Maximo can freely choose the target architecture, for example finite-state machines (for some discussions about alternative target architectures, see Appendix A). In this dissertation, I present my work on the role of the optimization algorithm in the context of automatic modular design.

1.1 Original contributions

This dissertation contains a number of original research contributions. In this section, I summarize each of the research contributions in the order in which they appear in this dissertation. For each contribution, I refer the reader to the chapter where the contribution is discussed and potentially the related scientific publications.

The main idea: I argue that offline optimization-based design has shown promising results in many missions but that the research composing the state of the art lacks a focus on what is arguably the central element of optimization-based design: the optimization algorithm. Based on this observation, I wished to explicitly study the role of optimization in the automatic offline design of control software for robot

swarms. Starting from `Chocolate`, an automatic modular design method that uses Iterated F-race, a model-free racing algorithm, as its optimization algorithm, I identified two additional classes of optimization algorithms that were of interest: local search algorithms and model-based optimization algorithms. I have tested and validated all proposed design methods, comparing them against other state-of-the-art automatic design methods. In total, I have performed over 3700 designs for 37 missions. Across all designs, I have used over $160\,\text{million}$ simulation runs to design control software and I have performed over $37\,000$ experiments in simulation and over 180 experiments on real robots to validate the performance of the generated control software. In particular, my work can be categorized into the following four contributions.

Review of the state of the art in swarm robotics: In Chapter 2, I provide an overview of the state of the art in optimization-based design of control software for swarm robotics. In particular, I focus on offline optimization-based design, including neuro-evolutionary design methods, automatic modular design methods, and imitation learning methods. The review of the literature is partially based on:

- **J. Kuckling** (2023). Recent trends in robot learning and evolution for swarm robotics. *Frontiers in Robotics and AI. To appear*.

A study of local search algorithms: I have developed design methods based on two local-search algorithms: iterative improvement and simulated annealing. The research studies presented in Chapter 4 led to new insights on the shape of the search space and thus the feasibility of employing local search algorithms in the automatic modular design of control software for robot swarms. The work on local-search based optimization algorithms is based on:

- **J. Kuckling**, T. Stützle, and M. Birattari (2020). Iterative improvement in the automatic modular design of robot swarms. *PeerJ Computer Science*, 6:e322.
- **J. Kuckling**[†], K. Ubeda Arriaza[†], and M. Birattari, (2020). AutoMoDe-IcePop: automatic modular design of control software for robot swarms using simulated annealing. In B. Bogaerts, et al. (Eds.), *Artificial Intelligence and Machine Learning: BNAIC 2019, BENELEARN 2019*, Communications in Computer and Information Science, vol. 1196, 3–17.

[†]The authors contributed equally to this work and should be considered co-first authors.

A study of model-based optimization algorithms: Furthermore, I developed two design methods based on SMAC2 and SMAC3, two model-based design algorithms. The research study presented in Chapter 5 led to new insights on how surrogate models can improve the quality of the fine-tuning of parameters. The work on model-based optimization algorithms is based on:

- **J. Kuckling**, H. H. Hoos, T. Stützle, and M. Birattari, (2023). Comparison of different optimization algorithms in the automatic modular design of control software for robot swarms. *To be submitted for journal publication*.

The vision: Lastly, I have proposed my perspective on optimization-based design of control software for robot swarms—see Chapter 6. I have identified three levels, defined by their conceptual approach to optimization. Furthermore, I have discussed the differences and similarities between optimization-based design of control software for robot swarms and related domains. This work is based on:

- **J. Kuckling**, H. H. Hoos, T. Stützle, and M. Birattari, (2023). Design of collective behaviors for robot swarms: A perspective on the optimization-based design of robot swarms. *To be submitted for journal publication*.

1.2 Further contributions

Additionally to the main contributions of this dissertation, I have also contributed to other research studies.

Behavior trees as an alternative control architecture: In Appendix A, I present several research studies that employ behavior trees instead of finite-state machines as the control architecture into which the modules are assembled. The results obtained from these studies provide insights on the feasibility of using behavior trees as a control architecture and the challenges of defining modules to be used with behavior trees. The work on behavior trees is based on:

- **J. Kuckling**[†], A. Ligot[†], D. Bozhinoski, and M. Birattari (2018). Behavior trees as a control architecture in the automatic modular design of robot swarms. In M. Dorigo, et al. (Eds.), *Swarm Intelligence: 11th International Conference, ANTS 2018*, Lecture Notes in Computer Science, vol. 11172, 30–43.

- A. Ligot[†], **J. Kuckling**[†], D. Bozhinoski, and M. Birattari (2020). Automatic modular design of robot swarms using behavior trees as a control architecture. *PeerJ Computer Science*, 6:e314.

- **J. Kuckling**, V. van Pelt, and M. Birattari (2022). AutoMoDe-Cedrata: automatic design of behavior trees for controlling a swarm of robots with communication capabilities. *SN Computer Science*, 3:136.

Search space considerations: Lastly, I performed several theoretical studies of the search space. In particular, I have studied the size of the search space for AutoMoDe-`Chocolate` and AutoMoDe-`Maple`, as well as proven that the neighborhood function defined for AutoMoDe-`Cherry` and AutoMoDe-`IcePop` is complete. The considerations of the search space are based on:

- **J. Kuckling**, A. Ligot, D. Bozhinoski, and M. Birattari (2018). Search space for AutoMoDe-Chocolate and AutoMoDe-Maple. Technical Report TR/IRIDIA/2018-012, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.
- **J. Kuckling**, T. Stützle, and M. Birattari (2020). Iterative improvement in the automatic modular design of robot swarms. *PeerJ Computer Science*, 6:e322.

1.3 Outline

The rest of this dissertation is structured as follows. In Chapter 2, I provide an overview of the state of the art in swarm robotics, with a focus on offline optimization-based design. In Chapter 3, I describe some general methodology that will be used throughout this dissertation. In Chapter 4, I present the design methods and research studies based on local search algorithms. Similarly, in Chapter 5, I describe my work on using model-based optimization algorithms. In Chapter 6, I describe my vision on optimization-based offline design and similarities and differences to related domains. Lastly, in Chapter 7, I conclude the work presented in this dissertation and provide an outlook on future work.

In Appendix A, I present my research on using behavior trees as an alternative control architecture in the automatic modular design of control software for robot swarms. In Appendix B, I present a theoretical consideration about the size of the search space and a theoretical proof of the completeness of the neighborhood used in Chapter 4.

Chapter 2

State of the art

A robot swarm is thought to be an autonomous, decentralized, self-organized, and highly redundant system (Şahin, 2005; Brambilla et al., 2013; Dorigo et al., 2014). It does not rely on any external infrastructure, nor on any other centralized entity, such as a single leader robot. The collective behavior of the swarm arises from the interactions of the robots among each other and the environment. For any role in the swarm, such as a specific specialization, multiple robots in the swarm possess the ability to perform the necessary actions for that role. As a result, the desired collective behavior does not depend on any single robot.

The individual robots are simple in terms of hardware and software, when compared to the collective behavior that should be achieved by the swarm. Indeed, the desired collective behavior is commonly a behavior that no single robot of the swarm could achieve on its own. Furthermore, the sensing and communication capabilities of each robot are local, and the robots do not have access to global information.

These characteristics of a robot swarm are deemed beneficial for the realization of systems that exhibit desirable properties, such as flexibility, scalability and fault tolerance (Beni, 2005). Flexibility is the ability of a system to react and adapt to different environments. Since robot swarms are autonomous and self-organized, they can adapt to different environments. Fault tolerance is the ability of a system to continue to function, even if parts of it fail. A fault tolerant system should be able to handle partial failures (such as sensor malfunctions on a robot) and complete failures (a robot ceasing all functioning). Obviously, each failure will lead to a degradation of performance, but in a system that is not fault tolerant, a failure might hinder the continued operation completely. Robot swarms promise fault tolerance due to two properties. As they are decentralized, there exists no single point of failure. Furthermore, the swarm is highly redundant, with each role in the swarm having multiple robots that could perform it. Thus, the failure of any individual robot will not prohibit the continued operation of the whole

swarm. Scalability means that the exhibited behavior is independent of the size of the system. Scalable systems can operate in principle in a similar fashion on tens, hundreds, or thousands of robots. In a swarm, the robots can only perceive and interact with their immediate neighborhood. Assuming that the swarm density remains approximately constant, the robots are not aware of the actual size of the swarm. Thus, more robots outside of the perception radius of any robot will not affect the behavior of this robot.

Swarm robotics differs from classical single- or multi-robot systems. In single-robot systems, the challenge usually lies in programming the robot so that it performs a complex behavior. For example, a robotic manipulator might need to solve kinematics to move efficiently, a drone might need to use computer vision to detect obstacles, or a legged robot might need to move through uneven terrain. Conversely, in swarm robotics, it is assumed that the individual robot is easy to program. The challenge lies in programming the robots in such a way that the resulting interactions give emergence to a desired collective behavior. Robot swarms can be seen as a subclass of multi-robot systems. Classical multi-robot systems, however, often allow the robots to access global information, for example, through global communication or a centralized control entity, or are not highly redundant, for example, with only a single robot being able to perform certain roles.

Historically, the field of swarm robotics took inspiration from swarms in nature and early works in swarm robotics have often focused on recreating natural behaviors with robots or on integrating robots into groups of animals (Garnier et al., 2005; Schmickl and Hamann, 2011; Campo et al., 2011). However, the field of swarm robotics expanded beyond its source of inspiration and has matured into a more general discipline. The discipline of swarm engineering focuses on the engineering of arbitrary collective behavior (Winfield et al., 2005a; Brambilla et al., 2013). In swarm engineering, the design problem is: given a mission specification—robotic platform, environment, desired collective behavior—create control software such that the robots perform the desired collective behavior in the target environment.

Thanks to the desirable properties, swarm robotics is considered a prominent approach to control large groups of autonomous robots (Rubenstein et al., 2014; Werfel et al., 2014; Mathews et al., 2017; Garattoni and Birattari, 2018; Slavkov et al., 2018; Yu et al., 2018; Li et al., 2019; Xie et al., 2019) and has been identified as one of the grand challenges of robotics (Yang et al., 2018). Furthermore, industrial and public applications of swarm robotics are expected to emerge within the next 10-15 years (Dorigo et al., 2020, 2021).

The remainder of this chapter provides an overview of the key areas of interest in swarm robotics. I do not aim to provide comprehensive surveys of the topics covered, rather I focus on those relevant to this dissertation. At the appropriate places, I refer the reader to more comprehensive surveys. In Section 2.1, I highlight

some relevant swarm behaviors. In Section 2.2, I present several noteworthy robotic platforms. In Section 2.3 and Section 2.4, I discuss the challenges in designing control software for robot swarms. In Section 2.5 and Section 2.6, I provide a more in-depth review of different classes of automatic design methods. In Section 2.7, I discuss the open issue of the reality gap and current approaches to address it. In Section 2.8, I present the literature on the design method of the AutoMoDe family. In Section 2.9, I give an introduction to optimization theory with a focus on automatic algorithm configuration.

2.1 Swarm behaviors

The behavior exhibited by the whole swarm is called *collective behavior*. Although these collective behaviors can be arbitrarily complex, they are often composed of more basic behavior blocks. In this section, I present a selection of the fundamental swarm behaviors that have been studied in the literature. For more detailed reviews of collective behaviors, see the works of Brambilla et al. (2013), Hamann (2018), and Schranz et al. (2020).

2.1.1 Aggregation

Aggregation (or clustering) can be considered one of the most fundamental swarm behaviors (Hamann, 2018). In aggregation, the task for the robots is to position themselves in the environment, in such a way that they are close to the other robots in the swarm. It can serve as a building block in more complex behaviors, as it allows the robots to gather and interact with each other. For example, aggregation is usually a precursor to collective motion. The task might be further constrained by defining an appropriate aggregation site, which the swarm needs to find. Furthermore, there might exist multiple aggregation sites and the swarm needs to aggregate on the one with the highest quality.

Aggregation behaviors can also be found in many animal swarms in nature, such as honeybees or cockroaches. As a result, several works study aggregation algorithms that are inspired by animal behaviors. Garnier et al. (2005) developed a collective aggregation behavior that was inspired by the behavior of cockroaches. They first studied and modeled the behavior of individual cockroaches, which they implemented in the individual behaviors for a swarm of robots. The robots performed correlated random walks and wall-following behaviors in the environment. They stopped randomly, with the probability depending on the number of already stopped neighbors. The results showed that the emergent collective behavior of the robots was similar to the behavior observed in cockroaches. Halloy et al. (2007) integrated a group of robots into a swarm of cockroaches. The hybrid swarm was

placed in an arena with two potential shelters. From prior experience, it was known that the cockroaches would prefer to aggregate under the darker shelter. In the first experiment, the robots behaved similarly to cockroaches and aggregated under the darker shelter. In a follow-up experiment, the authors changed the behavior of the robots to steer the swarm to aggregate under the lighter shelter. Schmickl and Hamann (2011) developed an aggregation algorithm based on the aggregation behavior observed in honeybees. Robots explored the environment at random. When encountering another robot, they stopped for a time proportional to the quality of their current location. The algorithm allowed a swarm of robots to aggregate at the global maximum of a gradient. Notably, the robots had no memory and could not discern the global optimum from local optima. Arvin et al. (2014) later extended the algorithm with a fuzzy-based approach.

Outside of bio-inspiration, aggregation behaviors have also been studied from an engineering perspective. Soysal and Şahin (2005) studied the process of a static aggregation behavior. They developed a finite-state machine that had robots exploring the environment. When the robots encountered another robot, they would stop moving. At every further time step, every robot would randomly decide to stay stopped or resume wandering, according to a fixed probability parameter. Gazi (2005) used control theory and potential fields to derive a control policy for a swarm. The robots were controlled by an artificial potential field defining attractive and repulsive forces among them. Trianni et al. (2003) engineered an aggregation behavior through the use of evolutionary swarm robotics (see Section 2.6.1) where the robots were controlled by an artificial neural network. With this approach, they could generate both static and dynamic aggregation behaviors. Gauci et al. (2014c) studied aggregation with minimal robots. The robots are memory-less and have a single binary sensor. During the design process, a mapping of the two input states to outputs for the wheels is learned. Despite the simplicity of robot and control software, the swarm was able to aggregate.

2.1.2 Dispersion

Dispersion can be seen as the dual problem of aggregation. Instead of having the swarm position the robots close to each other, in dispersion they need to spread out while remaining in contact. Often, dispersion is related to coverage, in which robots need to cover a maximum amount of space.

A main challenge in dispersion is to perceive the presence, direction and distance of other robots. Several authors, therefore, investigated the use of different technologies. Payton et al. (2001) used the signal strength of infrared communication to estimate the distance between robots. They developed a dispersion algorithm that was inspired by the principles of gas expansion and tested the algorithm on a swarm of 20 robots. Ludwig and Gini (2006) developed an approach that

uses wireless signal strength to estimate distances. Unlike infrared communication, wireless signal did not provide the direction of neighboring robots. Still, the swarm could disperse in the environment. Building on the work of Ludwig and Gini, Ugur et al. (2007) used the strength of wireless signals to coordinate the dispersion in a swarm of 25 robots. The authors optimize the threshold parameter of their control software to achieve maximum coverage of the mission environment.

Other authors focused on the engineering of dispersion behaviors. Duarte et al. (2016) evolved a dispersion behavior for a group of aquatic robots. Robots could compute their relative positions to each other using GPS. The authors used NEAT to evolve the neural networks controlling the robots. Özdemir et al. (2019) studied the emergence of dispersion and coverage in a swarm of minimal robots. The robots are memory-less and only have a binary line-of-sight sensor. The authors used an evolutionary algorithm to find the appropriate wheel velocities for each of the two possible sensor states.

2.1.3 Foraging

Foraging is another behavior that can be found in animals. In swarm robotics, foraging can be characterized as a search and retrieval task. The robots need to find and transport items from sources to sinks that often are also called nests.

Several authors investigated foraging with a focus on developing efficient search strategies. For example, Sugawara and Sano (1997) investigated a foraging scenario with 5 robots. They programmed the robots with two behaviors: searching and homing. Their results showed that the collaboration encoded in the searching behavior was beneficial to the performance of the swarm. Nouyan et al. (2009) developed a chain-based foraging algorithm. The robots formed a chain between the source and the nest. Using a color code, the robots indicate the direction of the chain. Other robots could then follow the chain to retrieve the items. Hoff et al. (2010) proposed two algorithms for foraging, one relying on virtual pheromones and another relying on communication networks. In the pheromone-based algorithm, robots deposit two kinds of pheromone. One that leads towards the sources and one that leads towards the nest. During the execution, some robots stopped their search and communicated the local value of pheromone to other robots. In the cardinality-based algorithm, robots can also either forage or be beacons. Instead of communicating pheromone values, the beacons transmitted their distance to the nest. The distance was expressed by the number of communication jumps between the beacon and the nest. Campo et al. (2010) developed an algorithm for path selection in foraging problems. The algorithm could discriminate sources according to path lengths and profitability. This allowed the swarm to forage from the highest quality source and let the swarm decide on the shortest path to reach sources of equal quality. A complex foraging task was tackled in the Swarmanoid

project (Dorigo et al., 2013). Three different groups of robots needed to cooperate to retrieve a book from a shelf. Notably, no group of robots possessed the ability to retrieve the book with the others.

Other authors have studied foraging as an application to task-allocation. The robots in the swarm could take on different roles and the swarm needed to find an appropriate equilibrium. Liu and Winfield (2010) manually designed a probabilistic finite-state machine for the foraging task. The robots were alternating between searching for food, depositing food, and resting in the home area. Castelló Ferrer et al. (2016) extended the approach and proposed an adaptive threshold algorithm. The adaptive threshold approach allowed the swarm to update the transition rates during the experiment. Pini et al. (2011) used a task partitioning approach to foraging. Instead of transporting the items all the way to the nest, robots had the choice of dropping items into a cache area. Other robots could collect the items from the cache area and transport them to the nest. The swarm then needed to dynamically allocate the tasks to avoid interference and maximize the number of retrieved items.

2.1.4 Collective decision making

Another essential swarm behavior is collective decision making. In collective decision making, the swarm must select one option out of multiple possible. The challenge lies in the fact that robots do not have access to global information. As a result, robots need to explore the environment and synchronize their beliefs. Collective decision making usually requires the swarm to reach a decision based on options of differing quality (where the objective is to decide on the option with the highest quality) or on options with the same quality (where the objective is to break the symmetry and converge on a single option).

Collective decision making often overlaps with other swarm behaviors, such as aggregation or foraging, where the swarm might need to reach a consensus on which option is most suitable for aggregation or foraging. Therefore, some works on aggregation and foraging could also be classified as collective decision making. Notably are the works of Halloy et al. (2007) and Pini et al. (2011). Halloy et al. steered the decision making process of cockroaches and made them aggregate in the shelter that was of relatively lower quality. In the case of Pini et al., the swarm had to reach a consensus on whether using an intermediate storage was more efficient than doing direct transport.

Campo et al. (2011) proposed a decision making algorithm inspired by the collective decision making observed in cockroaches and ants. Robots were leaving a resource, with a random probability depending on the density with which they were occupied. Their swarm could decide on the smallest resource that was still sufficiently large to accommodate the whole swarm. Hamann et al. (2012)

investigated the symmetry breaking effects that occur when a swarm of robots need to decide between two options of the same quality. Robots were controlled by the BEECLUST algorithm (Hamann et al., 2012) and had to aggregate on one of the two spots of equal quality.

Several researchers also investigated decision making as a process without a direct application in mind. Valentini et al. (2014) studied a simple decision making algorithm, in which robots either survey one of the two options or disseminate their opinion. The dissemination duration was proportional to the quality that the robots associated with their chosen option. The authors showed that, with increasing swarm size, this algorithm increased decision accuracy. In a follow-up work, Valentini et al. (2016) studied the trade-offs between accuracy and speed in reaching a consensus in a binary decision problem. They tested their algorithm on a swarm of 100 Kilobot robots. Ebert et al. (2017) studied a collective decision making scenario in which the swarm needed to reach consensus across multiple independent features. They combined single-feature decision making with a task allocation strategy that was determining which feature a particular robot was currently deciding about.

2.2 Robotic platforms

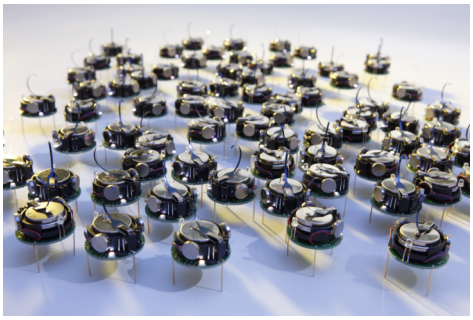
Swarm robotics research has been conducted on various robotics platforms (for example, see Figure 2.1). In the following, I present a few examples that have either found widespread application or have been used in notable experiments. In principle, robotic platforms used in swarm robotics can be categorized into three categories: ground-based mobile robots, aerial drones, and aquatic robots.

2.2.1 Ground-based mobile robots

Ground-based mobile robots are the most used class of robots in swarm robotics. They are characterized by the fact that they operate on the ground and can move of their own accord. Typically, they contain at least proximity sensors, accelerometers, gyroscopes and LEDs. Often, they also contain ground sensors, light sensors, microphones, and a camera which can be omnidirectional or directional. If not further specified, it is assumed that all robots have these sensors. Further sensors and actuators can be included according to the use cases foreseen for the robot. However, most robotic platforms allow to add external modules to customize the robots. Communication between robots is often implemented using either visual communication (e.g., LEDs), point-to-point communication (e.g., range-and-bearing), or radio communication (e.g., Wi-Fi, Bluetooth).

(a) e-puck (Mondada et al., 2009).

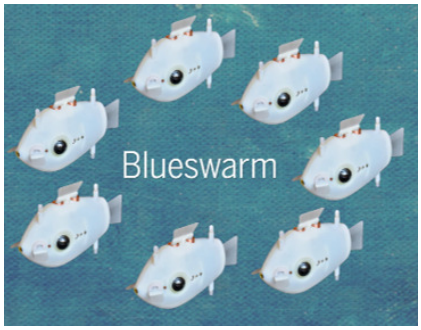
(b) foot-bot robots (Dorigo et al., 2013).



(c) Kilobot (Rubenstein et al., 2014).

(d) Thymio (Vitanza et al., 2019).

(e) Aquatic surface robot (Duarte et al., 2016).



(f) Blueswarm (Berlinger et al., 2021).

Figure 2.1: Examples of robot platforms used in the literature.

The Khepera robot was developed in 1992 (Mondada et al., 1997). It is a cylindrical robot with $5.5\,\mathrm{cm}$ in diameter that can move with a differential drive system. The base of the robot can be extended with application-specific turrets. Krieger et al. (2000) studied scalability and flexibility for a foraging swarm of up to 12 Khepera robots. The robots were extended with a turret that allowed them to grip and transport items and a radio communication module. Ijspeert et al. (2001) deployed a swarm of up to 6 Khepera robots in a collaborative stick-pulling experiment. The robots were outfitted with a gripper that allowed them to lift a stick, but it could only be fully pulled out its hole if two robots cooperated in the pulling process. Several iterations of the Khepera robot have been proposed, the latest being the Khepera IV (Soares et al., 2015). It is a cylindrical robot with $0.14\,\mathrm{m}$ of diameter, differential drive, and a Wi-Fi module for communication.

The s-bot (Mondada et al., 2003) is a cylindrical robot with a diameter of $0.12\,\mathrm{m}$ that uses a differential drive system based on a combination of tracks and wheels for locomotion. It can communicate with its peers using sound and colors, and to assemble into larger structures by connecting to other robots with a gripper. The s-bot has been used in various missions. For example, Trianni et al. (2003) evolved aggregation behaviors for a swarm of up to 20 s-bot robots. Christensen and Dorigo (2006) evolved a combined phototaxis and hole-avoidance behavior for a swarm of up to 16 physically connected s-bot robots. Nouyan et al. (2009) programmed a swarm of up to 10 s-bot robots to perform chain formation for an item retrieval task.

The e-puck is a robot designed for educational and research purposes (Mondada et al., 2009). It is a cylindrical robot with a diameter of $0.14\,\mathrm{m}$ that moves through a differential drive system and can communicate with its peers through its LEDs. It can be extended through several modules (Gutiérrez et al., 2009; École polytechnique fédérale de Lausanne, 2010; Jones et al., 2018a; Allen et al., 2020). The e-puck2 is an evolution of the e-puck with the same capabilities but a modernized set of sensors and a newly added Wi-Fi module. Similar to other robotic platforms, the e-puck has been used in various missions. For example, Silva et al. (2015) developed an online distributed version of NEAT (Stanley and Miikkulainen, 2002) that designed control software in an aggregation mission for swarms of up to 30 e-puck-like robots. Francesca et al. (2014) developed AutoMoDe-`Vanilla`, an automatic modular design method that designed control software for a swarm of 20 e-puck robots in missions, such as foraging or aggregation. The robots were augmented with a range-and-bearing board, allowing communication among them. Jones et al. (2019) used a swarm of 9 e-puck robots extended with a GPU-based computation module to run an online evolutionary algorithm to solve a collective transport task. Hasselmann et al. (2021) compared the performance and reality gap of nine neuro-evolutionary design methods for a swarm of 20 e-puck robots. They considered five missions, namely aggregation, homing, foraging, gate passing, and

shelter.

The Swarmanoid (Dorigo et al., 2013) is a heterogeneous swarm composed of three different types of robots: foot-bots, hand-bots and eye-bots. The foot-bots are cylindrical robots with $0.17\,\mathrm{m}$ in diameter. They can move using a differential drive system, communicate using a range-and-bearing board and can use their gripper to attach to other foot-bots or hand-bots. The hand-bots can climb vertical structures and grip items. They cannot move horizontally out of their own accord, but the foot-bots can attach to a hand-bot and carry it. The eye-bots are aerial drones that can fly and observe the environment from above. The Swarmanoid has been employed to retrieve a book from a shelf in an unknown environment (Dorigo et al., 2013).

The Kilobot is a minimalistic robot platform (Rubenstein et al., 2014). It is a three-legged cylindrical robot with a diameter of $0.033\,\mathrm{m}$. It contains one ambient light sensor and one infrared transmitter and receiver for communication. Unlike other platforms, it does not have proximity sensor, accelerometers or other sensors. Still, it is possible to address various missions even with this simple robotic platform. Rubenstein et al. (2014) designed a swarm of up to 1024 Kilobot robots that could successfully self-assemble into various shapes. Talamali et al. (2021) studied the decision making with constraint communication in a swarm of 50 Kilobot robots.

The Thymio is another educational platform. It is approximately square, with a rounded front, and a side length of approximately $0.11\,\mathrm{m}$ (Vitanza et al., 2019). The robot is equipped with a differential drive system for movement and provides communication capabilities through its Wi-Fi module. It also includes a USB port through which it can be extended. Kaiser and Hamann (2022) designed control software for a swarm of four Thymio robots in an object removal mission.

DOTS is a test bed for industrial swarm applications (Jones et al., 2022). The DOTS robots are cylindrical with a diameter of $0.25\,\mathrm{m}$ and can move with holonomic drive. Additionally, they have a lifting platform on top that can be used to transport items.

2.2.2 Aerial drones

Research on swarms of aerial drones usually does not assume a specific robotic platform. Instead, they often target commercially available off-the-shelf solutions. However, a few notable platforms have been developed outside of the commercial sector.

The Distributed Flight Array is a multi-rotor vehicle that is composed of several individual modules (Oung and D'Andrea, 2011). Each individual module can autonomously lift itself in the air, but only in combination with other modules can it achieve stable flight. The eye-bots have been developed as part of the

Swarmanoid project (Dorigo et al., 2013). Together with the foot-bots and hand-bots, they formed a heterogeneous swarm. Their role in the swarm was to fly above the other robots and observe the environment from there. The S-drone is a robotic platform specifically designed for the use in robot swarms (Oğuz et al., 2022). It has several sensors, such as four cameras, optical flow sensors, a Lidar, and several time-of-flight sensors for obstacle avoidance.

A minimalistic alternative to regular aerial drones are micro air vehicles. These small drones offer only a limited selection of sensors but usually are less costly than a regular drone. The Crazyflie (Giernacki et al., 2017), for example, is a low-cost micro air vehicle designed for education and research. The Crazyswarm (Preiss et al., 2017) consists of 49 Crazyflie drones that perform coordinated formation flight. As the drones do not support complex onboard operations, online path planning was performed by a centralized control station outside of the swarm.

2.2.3 Aquatic robots

Aquatic robots in swarms often take one of three forms: Surface robots (often as mono-hull boats), propellor-driven underwater robots, or bio-inspired robots. Schmickl et al. (2011) developed CoCoRo, a heterogeneous swarm of underwater robots. The swarm consists of autonomous underwater vehicles searching the seabed for targets and base stations that act as chargers and virtual fences for the autonomous underwater vehicles. The base station floats on the surface and the underwater vehicles can move through the use of propellors. Both types of robots contain a variety of sensors to monitor the state of the swarm and the environment. Duarte et al. (2016) developed a swarm of 10 autonomous aquatic surface robots. The robots can move autonomously across the water surface and communicate via a Wi-Fi ad hoc network. They only contain a minimal set of sensors, namely GPS and a compass. Berlinger et al. (2021) developed Blueswarm, a swarm of seven fish-like robots. The robots move through the use of fins and can communicate through LEDs located on their bodies.

2.3 Microscopic and macroscopic models

In swarm robotics, the main challenge is predicting the global emergent behavior from the behavior of the individual robots, and vice versa. Martinoli et al. (1999) first studied the relation between the microscopic and the macroscopic level in the cooperative stick-pulling experiment. The microscopic level is concerned with the local behavior of the individual robots, whereas the macroscopic level describes the global behavior of the swarm as a complete system. Each level can be modeled independently, resulting in microscopic models describing the dynamics

encountered by the individual robots (but often unaware of the global state of the swarm) and macroscopic models describing the global properties of the swarm (but often abstracting away local properties such as position or internal states of the individual robots).

In a preliminary work, Martinoli et al. (1999) developed a probabilistic microscopic model for an item retrieval and clustering task. The authors define the probabilities for different events that the robot can encounter and use the probabilistic model to create a probabilistic simulation and compare their results with physics-based simulations and real robot experiments. Similarly, Ijspeert et al. (2001) developed a probabilistic microscopic model for a collaborative stick-pulling experiment. They compared the probabilistic simulations generated by their model with physics-based simulations and real robot experiments to study the effects of different parameters on the efficiency of collaboration. Lerman et al. (2001) used rate equations to create a macroscopic model for the same stick-pulling task. The authors developed a macroscopic analytical model that allowed to study the long-term time independent behavior of the swarm. Notably, the time complexity of solving the analytical model does not depend on the number of robots in the swarm, unlike for physics-based or probabilistic simulations. Martinoli et al. (2004) formalized the insights gained from the stick pulling experiments and proposed a methodology for modeling robot swarms in missions non-spatial metrics.

Another well-studied mission is aggregation. Winfield et al. (2005b) used linear temporal logic to create a microscopic model of the aggregation behavior of the swarm. The authors suggested that in the future logic and theorem proving might be techniques that could be used to prove the existence of global properties of the swarm. Soysal and Şahin (2007) developed a macroscopic model describing the size of the largest cluster in an aggregation task. They modeled the aggregation as a Markov chain and derived the steady state behavior of the system. The predictions of the macroscopic model were verified against physics-based simulations. Halloy et al. (2007) investigated the aggregation behavior of cockroaches in the presence of robots. It was previously observed that cockroaches aggregate in groups under dark shelters. The authors then introduced robots to steer the aggregation behavior of cockroaches. They developed a macroscopic model that described the distribution of cockroaches and robots among the shelters. Hamann et al. (2008) created two macroscopic models based on ODEs and PDEs respectively for a spatially explicit aggregation behavior. They validated the two models using experiments with real robots.

More recently, much work has been devoted to understanding and modeling the dynamics of collective decision making in swarm robotics. For example, Massink et al. (2013) used Bio-PEPA, a process algebra language, to model a collective decision-making behavior. Based on the Bio-PEPA model, the author performed various analyses, such as a stochastic Gillespie simulation, Monte-Carlo

simulations, and fluid flow analysis. Hamann et al. (2014) modeled a collective decision making process using chemical reaction networks. Based on rules for local interactions (modeled as chemical reactions), they derive a set of equations predicting the proportion of robots in each state. They validate their models with stochastic and physics-based simulations. Based on the results of Hamann et al., Vigelius et al. (2014) further studied the dynamics in collective decision making. They derived a macroscopic model using chemical reaction networks. Valentini et al. (2016) investigated the influence of the neighborhood size on collective decision making. The authors created a macroscopic model of the system using ordinary differential equations. The model was validated against real robot experiments. Reina et al. (2018) considered spatiality in collective decision making. The authors developed a macroscopic model in the form of stochastic differential equations from which they derived an appropriate parametrization of the corresponding microscopic model. The model was validated using both physics-based simulations and real robot experiments. De Masi et al. (2021) investigated how the distribution of different roles influences the outcomes of a collective decision making scenario. They proposed a macroscopic model based on ordinary differential equations to describe the state of the swarm. The model is validated against experiments done in simulation and with real robots.

Several other missions have been studied as well. For example, Lerman and Galstyan (2002) developed a macroscopic model for a foraging swarm. Using rate equations, the authors studied the effects of interference with increasing swarm sizes in a bounded arena. Lerman et al. (2006) derived a macroscopic model of task allocation from a microscopic model. They first modeled the dynamics of task allocation in a single robot as a stochastic equation. From this equation, they derive an equation that describes the distribution of opinions in the swarm. They tested the macroscopic model against physics-based simulations. Correll and Martinoli (2007) formalized a microscopic model for distributed graph coverage. They investigated the degree of deviation between the prediction of a perfect model and a noisy simulation. Based on these results, they estimated the value of an additional parameter in the model that tries to encapsulate the effects of imperfect localization. Hamann and Wörn (2008) developed spatial models for tasks related to robot motion. They used the Langevin equation and the Fokker-Planck equation to derive a macroscopic model for a collective motion task and a collective perception task. Matthey et al. (2009) used chemical reaction networks and ODEs to create macroscopic models with various levels of abstractions for the behavior of a swarm in a puzzle assembly task. They use the models to predict optimal values for a selected parameter and validated the models and parameter choice in a physics-based simulation. Prorok et al. (2011) developed several microscopic and macroscopic models for describing the random walk behavior of a group of robots. The models are built around various techniques, such as Markov

chains, diffusion models, and difference equations.

2.4 Design methods

As shown in the previous section, it is difficult to match the dynamics of the collective behavior and the dynamics of the individual behaviors in a swarm. This poses a challenge for designing control software for robot swarms, as the desired collective behavior is usually expressed at the collective level, but the software needs to be generated at the individual level. However, there exist several techniques to design control software. Formal design (see Section 2.4.1) makes use of known micro-macro links. When the desired collective behavior is specified, it can be translated into control software for the individual robots. In manual design (see Section 2.4.2), a human designer creates the control software for the individual robots. This is often done through a trial-and-error process, but several software engineering techniques and patterns have been proposed to help the human designer. In optimization-based design (see Section 2.4.3), the design problem is cast as an optimization problem. Instead of designing control software that results in a desired collective behavior, an optimization algorithm searches the space of possible instances of control software for one that maximizes a given objective function.

2.4.1 Formal design

In missions where a formal micro-macro link exists, it can be used to automatically derive the control software for the individual robots. When deriving the macroscopic model, it is often assumed that the behavior of the robots on the individual level is fixed (except for some free parameters). As a result, formal design exploiting micro-macro links is only capable of designing control software for tasks that already are understood on the individual level. Yet, formal design allows to analyze and find optimal sets of the free parameters.

For example, Liu and Winfield (2010) modeled the dynamics of a foraging behavior using rate equations. They used a genetic algorithm to tune the parameters of their adaptation algorithm and used their macroscopic model to quickly evaluate sets of parameters. Berman et al. (2009) designed an optimal transition matrix for a task allocation problem. The robots were running a control software that allowed them to switch between different tasks with variable probabilities. The authors used Metropolis optimization (Landau and Binder, 2014) to compute the optimal transition probabilities for a given initial distribution of the swarm. Prorok et al. (2017) extended this approach to task allocation with robot specializations. They first developed the macroscopic model and then used gradient descent-based

optimization to compute an optimal transition matrix. In another work, Berman et al. (2011) developed a macroscopic model for a pollination task. They used Monte Carlo simulations to find the optimal value for a parameter of the individual control software.

An alternative approach to formal design is automatic rule synthesis. Rather than just finding optimal values for free parameters, automatic rule synthesis decomposes the desired collective behavior into rules that the individual robots must follow. Often, these rules are not directly turned into control software but embedded within a more complex instance of control software that interprets the rules and handles the actuation of the robot. For example, Yamins and Nagpal (2008) developed a global-to-local compiler for a pattern generation task. Based on the description of a desired final pattern, a rule set was automatically generated that allowed a swarm of randomly initialized robots to converge into the desired pattern. Werfel et al. (2014) followed a similar approach for a three-dimensional construction task. The desired shape of construction was used to automatically synthesize local construction rules that the robots could follow. Lopes et al. (2016) developed a design method based on supervisory control theory to control swarms with up to 600 robots. Their approach synthesizes control software derived from a formal language representation of the task and the capabilities of the robots.

2.4.2 Manual design

Manual design often follows a trial-and-error strategy (Francesca and Birattari, 2016). The human designer conceives an instance of control software and evaluates it. Depending on the observations made, the designer redesigns the control software and evaluates it again. The process terminates when the human designer is satisfied with the performance of its control software. While this approach has resulted in the creation of many, often complex, swarm behaviors (Brambilla et al., 2013), it is usually time-consuming and strongly depends on the expertise of the designer. Consequently, trial-and-error design cannot provide guarantees on the performance and is usually not reproducible.

Therefore, several authors proposed strategies for a more structured approach to manually designing control software for robot swarms. Kazadi (2009) developed physicomimetic (Spears and Spears, 2012) control software for a flocking task from model-independent requirements. In a first step, he described the behavior in a model-independent way. Then he applied different physicomimetic models in an attempt to find a model that fulfilled the established requirements. Brambilla et al. (2014) proposed property-driven design, a top-down design approach for swarm robotics based on model-checking. Their approach consists of four steps: first, the requirements are formally specified; second, macroscopic model derived from the requirements; third, this model is used to guide the implementation of the control

software in simulation; and finally, the control software is implemented on real robots. Like test-driven design, their approach iterates between implementing parts of the design and verifying them against prior established requirements. Reina et al. (2015) propose design patterns for the design of collective decision tasks. The design patterns allowed to design control software for a variety of decision making scenarios. Pinciroli and Beltrame (2016) developed Buzz, a programming language that allows to program swarms directly. It provides a library of several swarm-level action primitives that the user can use to program the swarm. The swarm-level program is then compiled into control software that will be executed by each robot locally. Marshall et al. (2019) developed an interactive modeling tool that allows users to create and analyze macroscopic models of their swarm behavior. To that end, users can specify a set of local interaction rules that are interpreted as chemical reaction networks. The tool then provides different analyses, for example, a bifurcation analysis of the corresponding global models.

2.4.3 Optimization-based design

An alternative to the manual design of control software is optimization-based design (Birattari et al., 2020). In optimization-based design, the design problem is reformulated as an optimization problem: given a mission specification—environment, robotic platform, and mission-specific performance measure, called the objective function—an optimization algorithm searches the space of possible instances of control software for the one that optimizes the objective function. The objective function, as part of the mission specification, is the mathematical description of the desired collective behavior. However, it can happen that the objective function is not well suited for the optimization process. In these cases, domain knowledge can be introduced to steer the design process. In the domain of neuro-evolution, these types of functions are called *fitness functions*.

Design methods in optimization-based design of control software for robot swarms can be further classified according to several criteria (Francesca and Birattari, 2016; Bredeche et al., 2018; Birattari et al., 2020). In semi-automatic design, a human designer remains in the loop during the design process. The designer monitors the design process and can intervene if the resulting instance of control software is not deemed satisfactory. For example, the designer might adjust the parameters of the optimization process and restart it after assessing the returned instance of control software. In a research setting, semi-automatic design allows to assess the feasibility of design methods. However, in practice, semi-automatic design exhibits similar drawbacks as manual design, especially the inability to reproduce experiments and the dependency on the skill of the designer. Conversely, fully automatic design of control software does not allow for human intervention beyond specifying the mission. Once the mission has been specified, the design

process runs without any further human intervention until it returns a final instance of control software. Fully automatic design can be considered a *one-shot* design process, whereas semi-automatic design might allow iterations.

Another major distinction is between online and offline design methods (Francesca and Birattari, 2016; Bredeche et al., 2018). Online methods design the control software while the swarm performs the assigned mission. However, the robots can only evaluate the quality of control software from the local information available to them. This restricts the class of missions that can be addressed by using online design. Offline methods run the design process in an environment other than the mission environment—e.g., in simulation or in a mock-up environment. Simulations offer several benefits over assessing the performance on real robots: (i) they allow for faster-than-real-time execution of experiments, (ii) they do not require access to robotic hardware, and (iii) they allow access to information, that would not be available without external infrastructure. However, the drawback is that the design process risks over-fitting certain idiosyncrasies of the simulation context—these inescapable differences between simulation and reality are called the reality gap (Brooks, 1992; Jakobi et al., 1995).

The following sections provide overviews over recent advances in optimization-based design. In Section 2.5, I provide an overview of online optimization-based design methods. In Section 2.6, I survey recent offline optimization/based design methods. In Section 2.7, I discuss recent advances to tackle the challenge of the reality gap.

2.5 Online optimization-based design

In online optimization-based design, the control software is generated while the robots are deployed in the target environment. Typically, each robot in the swarm executes part of a distributed evolutionary algorithm. These design approaches are called *embodied evolution*. In principle, other approaches, such as reinforcement learning, could also be used to design online learning methods. Yet, they have not seen application in the online optimization-based design of control software for swarm robotics.

Nevertheless, recent work (Heinerman et al., 2015; Silva et al., 2017; Bredeche and Fontbonne, 2021) has framed the concepts behind embodied evolution as a form of learning, sometimes calling it *social learning*. The authors argue that embodied evolution is conceptually closer to learning (see Section 2.6.3), as the robots update their control software while performing the mission. Yet, the most common technique to implement embodied evolution remains the application of evolutionary algorithms.

In embodied evolution, in general, every robot in the swarm is initialized with

and maintains its own set of genomes. From this genome set, each robot selects one genome and executes the control software encoded in it. Periodically, all robots exchange genomes among each other (which may be subjected to mutation or crossover operations) and they select a new genome to execute. Embodied evolution provides important advantages with respect to offline approaches (Watson et al., 2002). As evolution takes place directly in the mission environment, transferability is no concern. Besides, the distributed nature of the design process allows exploring different solutions in parallel. The parallelization of the design process allows to speed up the production of control software if compared with centralized online evolutionary methods.

Only a few studies have been conducted using embodied evolution in swarm robotics. For example, Bianco and Nolfi (2004) investigated an embodied evolutionary approach in which robots share their genomes when physically connecting to other robots. Prieto et al. (2010) used embodied evolution to program a swarm of e-puck robots in a cleaning task. Bredeche et al. (2012) investigated the adaptivity of open-ended evolution to changes in the environment. Silva et al. (2015) developed an online, distributed version of NEAT (Stanley and Miikkulainen, 2002) and used it to evolve control software in three missions. Jones et al. (2019) evolved behavior trees for a collective pushing task. Cambier et al. (2021) used an evolutionary language model to tune the parameters of a probabilistic aggregation controller. For more detailed surveys, including online evolution for single and multi-robot systems, see, for example, the works of Bredeche et al. (2018) and Francesca and Birattari (2016).

Embodied evolution still faces several challenges in the context of the design of control software for robot swarms. For example, robots need to execute not only their own control software but also the design process. This may not be feasible for robots with limited computational hardware. Furthermore, to conduct the evolutionary process, the swarm must operate for a relatively long time (as compared to the normal mission duration), posing more demand on batteries and increasing the likelihood of sensor or actuator failures. More importantly, the evolutionary process can only be achieved if the individuals of the swarm can assess the performance of their chosen genome—ideally this should be computed for the whole swarm, however, without further infrastructure this information is not directly available to the robots as they rely only on local perception.

Three main solutions have been proposed to address the aforementioned issue: open-ended evolution, decomposition and simulation-based assessment. In open-ended evolution, the design process is not driven by an explicit objective function. Instead, open-ended evolution ties the survival of an instance of control software to its ability to “reproduce”. Over time, instances of control software that successfully reproduce will replace instances of control software that cannot. Implicit selection pressure can be exerted by tying the chance to reproduce to

certain desired actions or outcomes (Bianco and Nolfi, 2004; Prieto et al., 2010; Bredeche et al., 2012). Bianco and Nolfi consider encounters between robots as opportunity for reproduction. As the task is self-assembly, this implicitly rewards instances of control software that manage to encounter and assemble with other robots. Another typical choice is to model the performance of the individual robots by energy levels: taking an action depletes the energy, but certain outcomes of the actions replenish it. While the robot is active (with available energy), its control software is periodically exchanged with neighboring robots. Once the energy is fully depleted, the instance of control software that was active in the robot is replaced by another one. Over time, instances of control software that are more successful at managing their energy level (by achieving the desired outcomes) will have more opportunities to spread to other robots, thus prevailing in the swarm and displacing less successful instances of control software. Like novelty search, open-ended evolution does not necessarily aim to generate a particular behavior, but rather for the spontaneous emergence of complex behaviors. As an alternative, a designer could manually decompose the objective function for the desired collective behavior into rewards for the actions (or their outcomes) of individual robots (Silva et al., 2015). Although viable, this decomposition is especially difficult in tasks that strictly require cooperation or only provide delayed rewards—e.g., taking an action does not immediately increase the fitness of the swarm, as it requires an appropriate second subsequent action to effectively increase the fitness. This decomposition is similar to the credit assignment problem encountered in robot learning. Recently, Jones et al. proposed an online evolutionary method in which robots performed simulations to evaluate the quality of genomes (Jones et al., 2019). This method allows the robots to estimate the performance of a genome as if it was deployed to the whole swarm—without the need for decomposing the objective function. However, assessing the performance in simulation might overestimate the degree of cooperation and coordination of the robots, as other members of the swarm might execute different instances of control software and not cooperate as expected.

2.6 Offline optimization-based design

In offline optimization-based design, the control software is generated before the robots are deployed in the target environment. Typically, the design process is performed in a centralized manner and evaluates the quality of instances of control software in simulation. In the context of swarm robotics, *evolutionary swarm robotics* (Trianni, 2008; Nolfi, 2021) is the most studied optimization-based design approach. Evolutionary swarm robotics has been used to create control software for robot swarms in a wide variety of mission such as foraging, collective transport,

or pattern formation (see also Section 2.1) (Brambilla et al., 2013; Schranz et al., 2020). Traditionally, evolutionary swarm robotics has relied on *neuro-evolution*—the control software in the form of an artificial neural network is optimized using a centralized evolutionary algorithm (see Section 2.6.1).

Other typical design approaches include *automatic modular design* (see Section 2.6.2) and *multi-agent reinforcement learning* (see Section 2.6.3). In automatic modular design, the control software is composed of modules that are assembled into more complex control architectures, such as finite-state machines or behavior trees. In multi-agent reinforcement learning, reinforcement learning techniques are used to design the instances of control software. While offline optimization-based design methods have demonstrated promising results in the past, they still face some important challenges that remain unsolved: notably, the generation of control software that is robust to the reality gap and the specification objective functions that can produce a desired collective behavior.

Several works have addressed possible solutions to these challenges of offline optimization-based design methods. Concerning the reality gap, various approaches, such as *transferability approaches* or the concept of *pseudo-reality*, have been proposed (see Section 2.7). The formal definition of an objective function requires the attention of an expert, knowledgeable in the mathematical representation of collective behaviors. A few researchers have therefore investigated the use of mission specifications different from an objective function. For example, Bozhinoski and Birattari (2022) used a domain specific language to translate natural language descriptions of swarm missions into formal specifications. Other researchers have investigated the use of imitation learning (see Section 2.6.4) or novelty search (see Section 2.6.5) to design collective behaviors without an explicitly defined objective function. Yet, these approaches still rely on an objective function during the design process. The designer is only relieved during the mission specification, by using these techniques as an "interface" through which they can provide the specification in an implicit form, which is then automatically transformed into the objective function used during the design.

Trianni et al. (2014) and Francesca and Birattari (2016) provide overviews of offline optimization-based design in the context of swarm robotics.

2.6.1 Neuro-evolution

The application of evolutionary robotics principles (Nolfi and Floreano, 2000) to swarm robotics is called *evolutionary swarm robotics* (Trianni, 2008; Nolfi, 2021). In evolutionary swarm robotics, the control software of the robots is generated through an artificial evolutionary process. Unless otherwise specified, the same generated control software is uploaded to each robot to be executed individually. The objective function is used to assess the quality of instances of control software,

and in a way, provides selection pressure to direct the optimization process. Poorly performing instances are discarded and the well-performing ones are *selected* to generate new instances through *recombination* and *mutation*.

Neuro-evolution is one of the earliest optimization-based design methods in swarm robotics (Quinn et al., 2003; Trianni et al., 2003; Dorigo et al., 2003). In neuro-evolutionary methods, an evolutionary algorithm designs control software in the form of neural networks. In this approach, neural networks are used as *black-box* controllers, and the search performed by the evolutionary algorithm does not require domain-specific heuristic information. For this reason, neuro-evolutionary design methods are expected to allow the design of control software with no domain knowledge. For a review of early neuro-evolutionary design methods, see Brambilla et al. (2013).

More recently, several authors have focused on systematically using neuro-evolution to design control software for various robotic platforms—mainly targeting those that could be possibly used in real-world deployments. For example, Trianni and Nolfi (2011) evolved a perceptron network to synchronize the movement of a swarm of s-bot robots. Duarte et al. (2016) used NEAT (Stanley and Miikkulainen, 2002) to design control software a swarm of aquatic robots performing tasks such as homing or dispersion. Gomes et al. (2019) generated control software for robot teams composed of aerial and ground robots in a foraging task. Hasselmann et al. (2021) compared NEAT (Stanley and Miikkulainen, 2002), xNES (Glasmachers et al., 2010) and CMA-ES (Hansen and Ostermeier, 2001) to generate control software for a swarm of e-puck robots (Mondada et al., 2009) in five different missions such as aggregation, homing, shelter, foraging, and gate passing. In a different research direction, researchers have investigated the minimal requirements to evolve specific collective behaviors. For example, Gauci et al. (2014a) evolved a recurrent neural network to perform aggregation. In their study, the authors tested their control software on robots with minimal capabilities: each robot had a single binary sensor that controlled the speed of its two wheels. Gauci et al. (2014b) used a similar approach to study the emergence of collective behaviors for robots with minimal capabilities. In their study, the robots only had a single line-of-sight sensor and could set their velocity based on the discrete readings of this sensor. The authors used CMA-ES (Hansen and Ostermeier, 2001) to optimize the mappings of the sensor to velocities in missions such as clustering (Gauci et al., 2014b), shepherding (Özdemir et al., 2017; Dosieah et al., 2022), decision making (Özdemir et al., 2018), and coverage (Özdemir et al., 2019). Ramos et al. (2019) evolved a flocking behavior for robots that only had one alignment sensor and four proximity sensors. Diggelen et al. (2022) evolved a gradient following behavior. Notably, the robots could perceive only the local value of the gradient, not its direction, and they could not communicate with other robots.

Neuro-evolutionary approaches have shown many promising results. Yet, two

main challenges remain in the field: fitness engineering and the reality gap. The first challenge is fitness engineering. It is well understood that some objective functions pose two challenges to the evolutionary process: *bootstrapping* and *deception* (Silva et al., 2016). The issue of bootstrapping arises when the objective function fails to apply meaningful selection pressure in low-performance regions of the search space. As a result, the design process explores the low-performance regions in an undirected manner and is unable to converge towards higher performance regions of the search space. The issue of deception describes the case in which the objective function contains easily reachable local optima. In this case, the design process can easily converge towards the local optima and will result in the generation of a suboptimal collective behavior. These two issues can usually be overcome by using a different function (*fitness function*) than the objective function during the design process. The fitness function can then be engineered to avoid the issues of deception and bootstrapping by introducing *a priori* knowledge into it (fitness engineering) (Trianni and Nolfi, 2011; Divband Soorati and Hamann, 2015; Silva et al., 2016). However, the necessity of *a priori* knowledge conditions the effectiveness of a neuro-evolutionary design method; as it will largely depend on the expertise of the designer of the objective function. The second challenge of neuro-evolution is the reality gap. The reality gap are the inescapable differences between the design and deployment environment, and often manifests in a performance drop when designing control software in simulation and assessing it on real robots. Yet, not all design methods are affected similarly by the reality gap, and it is therefore imperative to assess all offline optimization-based design methods not only in simulation but on real robots (Birattari et al., 2019). In the context of neuro-evolution, Hasselmann et al. (2021) investigated the effects of the reality gap on different neuro-evolutionary design methods. They showed that, without further mitigation strategies or mission-specific adaptations, sophisticated neuro-evolutionary design methods perform similarly poor in reality as a simple perceptron network.

2.6.2 Automatic modular design

Neuro-evolution enables, in practice, the design of control software without prior domain knowledge. Yet, in cases that domain knowledge is available, it might be incorporated into the design method to achieve better results. Instead of relying on artificial neural networks, automatic modular design methods generate control software that is composed of software modules that are assembled into a more complex control architecture—e.g., finite-state machines or behavior trees (Colledanchise and Ögren, 2018). Through the choice and implementation of these modules, domain knowledge can be incorporated into the design process.

Duarte et al. (2014) manually decomposed a complex object removal task into

simpler subtasks. They evolved continuous-time recurrent neural networks that were then assembled, in a modular way, into a hierarchical controller (according to the manual decomposition). Ferrante et al. (2015) used grammatical evolution to design control software for a foraging scenario with task allocation. They designed behavioral rules from basic behavioral and conditional modules. Hecker et al. (2012) used a genetic algorithm to optimize a finite-state machine that controls the behavior of robots in a foraging swarm. The authors pre-programmed an initial finite-state machine, which was inspired by the foraging behavior observed in ants. They used the genetic algorithm to optimize parameters of the finite-state machine that were not chosen at design time. Francesca et al. (2014) proposed AutoMoDe-`Vanilla`, an automatic modular design method that assembles finite-state machines out of a set of twelve handcrafted modules (see Section 2.8 for a more detailed description of `Vanilla` and other design methods of the AutoMoDe family). Besides finite-state machines, behavior trees have recently gained attention in the literature of automatic modular design. They offer several advantages over finite-state machines, like enhanced modularity and better human readability (Colledanchise and Ögren, 2018). Jones et al. (2018b) evolved behavior trees for a foraging swarm of Kilobot robots (Rubenstein et al., 2014). Neupane and Goodrich (2019) used grammatical evolution to design software for a swarm of 100 robots performing a foraging task.

Automatic modular design methods are an emerging field of research with promising prospects. Preliminary results indicate that they are a viable alternative to neuro-evolutionary design methods, with comparable performance and better transferability between simulation and real robots. However, this advantage comes at the cost of devoting effort to specify the modules. An artificial neural network can map all possible sensory inputs to all possible actuator outputs. As a result, neuro-evolutionary design methods can be used to design control software to perform any mission that is within the capabilities of the robots. In the case of automatic modular design, a human designer must manually implement the modules. The choice of modules implicitly restricts the space of possible missions that can be addressed by an automatic modular design method (Garzón Ramos and Birattari, 2020). If the set of modules is too limited, the design method would only produce satisfactory results for the mission it was conceived for, and the design space might not contain well-performing instances of control software for other missions. In other words, the design method will underperform in most cases. In this situation, the underperforming method can be accepted as it is or it will become necessary to develop a new design method—which ultimately turns into a manual design method rather than an automatic one. An important question to be addressed is, therefore, how to develop general automatic modular design methods that still remain robust to the reality gap?

2.6.3 Multi-agent reinforcement learning

Reinforcement learning is a method for producing control software in which an agent attempts to learn a policy that encodes the set of optimal actions in a dynamic environment (Kaelbling et al., 1996). Classically, reinforcement learning only considers a single agent interacting with the environment. In this case, the system is then often modelled as a Markov decision process. As robot swarms are composed of several individuals, they are usually modelled as *multi-agent reinforcement learning* problems. In multi-agent reinforcement learning methods, all members of the swarm typically act independently. For reviews of reinforcement learning in the single and multi-robot domain, see the work of Kober et al. (2013) and Zhao et al. (2020).

Although multi-agent reinforcement learning has been largely studied in the literature, it has seen little application in swarm robotics so far. The first application of reinforcement learning in a swarm robotics scenario is possibly the one of Matarić. Matarić (1997) studied reinforcement learning with a swarm of 4 robots that perform a foraging mission. In a follow-up work, Matarić (1997) introduced robot communication in the swarm to synchronize rewards between the robots. More recently, Hüttenrauch et al. (2019) used deep reinforcement learning to generate control software for a swarm of virtual agents. Bloom et al. (2022) investigated the use of four deep reinforcement learning techniques in a collective transport experiment.

The application of multi-agent reinforcement learning poses several challenges that still hinder its application to swarm robotics. A first challenge arises from the fact that, in swarm robotics, the desired behavior is usually expressed at the collective level, whereas the learning must happen at the individual level. Thus, when designing control software using reinforcement learning, the mission designer needs to decompose the reward function of the whole swarm into rewards that can be assigned for individual contributions. This problem is also known as *spatial credit assignment*. To this date, no generally applicable methodology exists to address this problem and most works use manual credit assignment (Matarić, 1998; Hüttenrauch et al., 2019; Bloom et al., 2022).

Another important issue is the representation of the state and action spaces in the learning process. Typically, a multi-agent reinforcement learning uses joint action and state spaces, which are concatenated over the individual action and state spaces of each individual agent. These joint spaces, however, suffer heavily from the curse of dimensionality, as they scale poorly both in the size of the individual spaces and in the number of agents. Consequently, addressing large swarm sizes is infeasible in practice. Furthermore, the joint space is not observable by any individual agent, due to the locality of information in a robot swarm. In this sense, the problem of multi-agent reinforcement learning for swarms is more correctly

modelled by a partially observable Markov decision process (Kaelbling et al., 1996). In the literature, two techniques have been mostly used to overcome the partial observability: reducing the joint action and state space to those that are pertinent to a single robot (Hüttenrauch et al., 2019; Bloom et al., 2022); or sharing information to synchronize the state beliefs of all members of the swarm (Matarić, 1998). However, the policy is executed decentralized on each individual robot.

2.6.4 Imitation Learning

A research field in reinforcement learning that has become of interest for swarm robotics researchers is imitation learning (Osa et al., 2018). In imitation learning, the reward function is assumed to be unknown. Instead, the learning process has access to demonstrations of the desired behavior. The agents attempt to learn a policy that results in a behavior that is similar to the behavior that has been provided in a demonstration.

Within the research on imitation learning and swarm robotics, Li et al. (2016) proposed Turing learning for swarm systems. Inspired by the Turing test, the system learns two programs. A first program controls the robots in the swarm, whereas the second program attempts to distinguish between trajectories from the originally demonstrated behavior and trajectories from the behaviors that are being generated through learning. Šošić et al. (2017) used inverse reinforcement learning to learn the behavior of two predefined particle models. Using SwarmMDP (a variant of decentralized, partially observable Markov decision processes), they reduced the multi-agent reinforcement learning problem to a single-agent problem. Alharthi et al. (2022) used video recordings of simulated robots to learn a behavior tree corresponding to the demonstrated collective behavior. Gharbi et al. (2023) used apprenticeship learning (Abbeel and Ng, 2004) to learn collective behaviors from demonstrations of desired spatial organizations for a swarm.

Design methods based on imitation learning that are being currently developed face two challenges. The first challenge is that existing methods typically require detailed demonstrations to produce their corresponding control software. The more detailed the demonstrations, the easier it is to imitate them. Most work on imitation learning in swarm robotics uses an already available behavior to generate trajectories that must be learned again by the swarm (Li et al., 2016; Šošić et al., 2017; Alharthi et al., 2022). The obvious drawback of this approach is that it is only suitable for cases in which an implementation of the desired collective behavior already exists. Alternatively, other approaches have focused on only demonstrating a few key elements of the collective behavior, instead of a full trajectory (Gharbi et al., 2023). The second challenge is that there is no well-established method to measure the similarity between a demonstrated behavior and a generated one.

2.6.5 Other approaches

As discussed previously, a major issue in evolutionary swarm robotics is the formal definition of the objective function. Some recent studies focus on the application of novelty search in swarm robotics. Instead of optimizing a mission-specific performance measure, novelty search generates a set of behaviorally diverse instances of control software (Lehman and Stanley, 2011). Gomes et al. (2013) used novelty search to generate aggregation and resource sharing behaviors in a swarm. Additionally, the authors combined the novelty metric with a performance metric to overcome limitations where novelty search could not escape large, low-performance regions of the search space. In a follow-up work, Gomes et al. (2017) applied novelty search to co-evolutionary problems. Gomes and Christensen (2018) also investigated how to generate task-agnostic behavior repertoires using novelty search. Hasselmann et al. (2023) proposed AutoMoDe-`Nata`, an automatic modular design method that uses novelty search to create basic behavioral modules, which then are combined into probabilistic finite-state machines. See Hasselmann (2023) for an overview of novelty search-based design methods in swarm robotics.

Outside of novelty search, Trianni and López-Ibáñez (2015) investigated the use of an evolutionary multi-objective optimization algorithm in a strictly collaborative mission. Next to the (singular) objective of the mission, the authors specified a secondary auxiliary objective to overcome the convergence to certain sub-optimal behaviors—although the auxiliary conflicted with the main objective. They showed that multi-objective optimization indeed avoided premature convergence, and that properly chosen auxiliary objectives have the potential to overcome the bootstrap problem.

Kaiser and Hamann (2019) investigated an approach named "minimizing surprise". Inspired by the *free energy principle* (Friston, 2010), Hamann (2014) used offline evolution to generate control software in the form of two neural networks, a prediction network and an action network. The action network controlled the robot, whereas the prediction network predicted the next sensor state. The design process aimed to minimize the prediction error. Results showed that, despite not selecting for swarm behaviors, basic self-organizing collective behaviors emerged during the design process. Kaiser and Hamann (2019) extended their work and proposed a system to systematically engineer self-organizing assembly behaviors using the "minimizing surprise" approach.

2.7 Reality gap

One of the biggest remaining challenges in optimization-based design remains the reality gap, also called the sim-to-real problem (Francesca and Birattari, 2016;

Ligot and Birattari, 2020). Traditionally, the reality gap is understood to occur when the simulation environment is too simplistic. Recent research, however, challenges this "complexity understanding" (Ligot and Birattari, 2020, 2022). For example, Ligot and Birattari (2020) reproduced the effects of the reality gap in simulation-only experiments. They developed two simulation models and designed control software in one model (taking on the role of the design context) and assessed it in the other (taking on the role of a pseudo-reality context). The effects of the reality gap became apparent regardless of the choice of design and pseudo-reality model. Their results showed that the "complexity understanding" does not sufficiently explain the occurrence of the reality gap. In a follow-up work, Ligot and Birattari (2022) systematically studied the prediction error of different evaluation strategies, such as evaluation in the design context, evaluation in one pseudo-reality context, and evaluation in many pseudo-reality contexts.

Francesca et al. (2014) argued that the reality gap did not occur due to too simplistic simulations, but because of inherent differences in the simulation context that are not present in reality—see also Francesca (2017). A sufficiently expressive design method will then invariably exploit these idiosyncrasies of the design context. When assessed in reality, these idiosyncrasies are no longer present, and the control software suffers from the reality gap. They argued the tendency to exploit the idiosyncrasies of the design context can be seen as akin to the problem of overfitting in supervised machine learning (Geman et al., 1992). Overfitting happens when the learning process models relationships between inputs and outputs that exist in the training set, but not in the underlying distribution from which the training set was sampled. When evaluating the learned model on a new dataset (such as the test or validation set), the model will fail to correctly model the relationship between input and output. It is said that the learning process modeled the noise in the training data and failed to generalize. The generalization error is typically understood to be composed of two elements: the variance error and the bias error (Geman et al., 1992). The variance error is caused by a hypersensitivity to the training data and any noise contained therein (overfitting). The bias error stems from the inability to capture the relations present in the training data (underfitting). It has been shown that, for artificial neural networks, their low bias (as universal function approximators) entails a high variance. Typically, it is therefore necessary to find a trade-off between bias and variance. See Section 2.8 for an in-depth discussion of the proof-of-concept design method proposed by Francesca et al. (2014) to introduce bias and other design methods of the AutoMoDe family that are built around it.

Several other approaches have been proposed to reduce the effects of the reality gap. For example, *system identification* can be used to develop more realistic simulators that intend to minimize the differences between simulation and reality (Bongard and Lipson, 2004; Zhao et al., 2020). Koos et al. (2013)

proposed the *transferability approach*, in which the transferability of generated control software is periodically assessed during the design process. The design method will therefore solve a multi-objective optimization problem, in which both the performance in simulation and in reality are considered. For an in-depth overview of approaches to the tackle the reality gap in swarm robotics, see Ligot (2023).

2.8 AutoMoDe

Francesca et al. (2014) postulated that the reality gap can be seen akin to the bias-variance trade-off in machine learning. Design methods with low bias can generate control software that represents any relationship between sensor input and actuator output. This allows these design methods to generate control software that (in principle) has the potential to act (near) optimally in all circumstances. However, they also show high variance and tend to overfit the idiosyncrasies of the design context instead of realizing the desired relationships. Francesca et al. proposed to reduce the variance of the design method by allowing for bias. By restricting the possible relationships between sensor readings and actuator outputs that the design method can produce, control software can be generated that does not overfit the idiosyncrasies of the design context and therefore transfers well to reality.

To validate their hypothesis, they developed AutoMoDe-`Vanilla`, an automatic modular design method (Francesca et al., 2014). Automatic modular design methods are composed of several elements. First, they design control software for a given robotic platform. In order to allow comparison on equal footing with other design methods (e.g., neuro-evolutionary design methods), the actual robotic platform is abstracted through a *reference model*, which provides a formalization of the available sensors and actuators. Based on the reference model, a set of *behavioral and conditional modules* is defined. These modules form the most important way to restrict the variance, as they define and restrict how the robot can react to its sensor readings. During the design process, the modules will be assembled and combined into more advanced structures, called the *target architecture*. By defining additional constraints on the target architecture, the variance can be further restricted. The last element of an automatic modular design method is the *optimization algorithm*. This algorithm combines the modules into the target architecture and fine tunes any parameters that the modules might have. Francesca et al. compared the performance of `Vanilla` with `EvoStick`, a yardstick implementation of neuro-evolutionary swarm robotics. As expected, `EvoStick` outperformed `Vanilla` in simulation, testament to its small bias. However, when assessed in reality, `EvoStick` was outperformed by `Vanilla`. This indicates

that the reduced variance of Vanilla indeed was beneficial to the transferability of the generated control software. Yet, human designers designing control software with the same constraints as Vanilla were able to outperform the automatic design (Francesca et al., 2015).

In a follow-up work, Francesca et al. (2015) developed AutoMoDe-Chocolate (for a detailed description of the design method, see Section 3.4). Chocolate is identical to Vanilla in all regards, except for the optimization algorithm, where Chocolate employs Iterated F-race instead of F-race. That is, Chocolate also assembles finite-state machines (under the same constraints) from the same set of behavioral and conditional modules. Francesca et al. compared the performance of control software generated by Chocolate with that generated by Vanilla, EvoStick and control software generated by human designers. The results obtained in five missions showed that Chocolate outperformed both Vanilla and the human designers, while remaining robust to the reality gap.

Hasselmann et al. (2021) showed that rank inversion is not only an artifact of EvoStick. The authors compared several design methods based on state-of-the-art neuro-evolutionary methods, such as NEAT (Stanley and Miikkulainen, 2002), xNES (Glasmachers et al., 2010) or CMA-ES (Hansen and Ostermeier, 2001). All neuro-evolutionary methods considered suffered similarly from the reality gap and exhibited rank inversions when compared to Chocolate. Furthermore, the design methods based on more advanced algorithms did not offer practical advantages over simpler implementations, such as EvoStick.

Based on Chocolate, several variants (also called AutoMoDe flavors) were developed to investigate different elements of the design process, such as different sets of manually or automatically crafted modules (Hasselmann and Birattari, 2020; Garzón Ramos and Birattari, 2020; Mendiburu et al., 2022; Spaey et al., 2020; Ligot et al., 2020a), and hardware-software co-design (Salman et al., 2019). However, these design methods all take the optimization algorithm for granted. Only two other works have addressed the choice of the optimization algorithm. When Francesca et al. (2015) proposed Chocolate, they changed the optimization algorithm from F-race to Iterated F-race. Cambier and Ferrante (2022) proposed AutoMoDe-Pomodoro, a class of automatic modular design methods that use evolutionary algorithms as optimization algorithm. The authors proposed three design methods that use different encodings of the finite-state machines and different evolutionary algorithms. Their results show that the design methods based on evolutionary algorithms achieve similar results to those achieved by Chocolate in simulation.

Notably, neither work aims to address questions on the role of optimization. The introduction of Chocolate aimed at outperforming manual design by using a more sophisticated optimization algorithm. Pomodoro aimed to investigate different encodings of finite-state machines. Neither work addresses questions

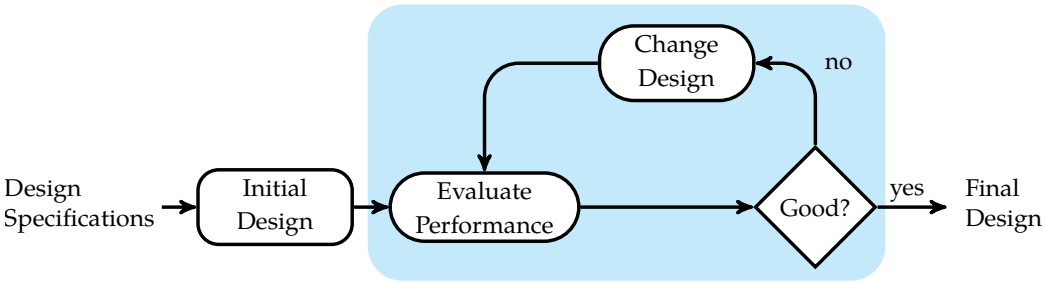


Image originally published by Kochenderfer and Wheeler (2019).

Figure 2.2: An overview of a general optimization process. The blue part denotes the role that an optimization algorithm plays in the process.

like what causes the observed differences in performance between two different optimization algorithms.

2.9 Optimization

A typical optimization process tries to find a *candidate solution* that is optimal with respect to some measure of quality. It can be seen as a repetition of two steps, as seen in Figure 2.2 (Kochenderfer and Wheeler, 2019). Before the optimization process enters its loop, the optimization problem needs to be formally specified. This step is often done manually by a human expert. The problem specification typically describes two elements: the representation of candidate solutions and a formal specification of the quality measure. The quality measure is often a mathematical function f mapping candidate solutions to numerical values. Depending on if it should be minimized or maximized, it is sometimes called a cost or utility function, respectively. More generally, the term *objective function* is used, encompassing both cost and utility functions. The representation of a candidate solution usually consists of a set of variables, which the optimization process can manipulate, and a set of constraints that describe conditions for a candidate solution to be valid. A candidate solution fulfilling all constraints is called feasible. The set of all feasible candidate solutions is denoted as C. The feasible candidate solution that optimizes the objective function is called the optimal solution to the optimization problem.

In the next step of the optimization process, one or multiple initial candidate solutions are generated from the problem specification. These candidate solutions are then evaluated on the quality measure. If the quality of one of the candidate solutions was sufficiently good (either optimal or close to an optimal solution), the optimization process terminates, and the candidate solution is returned. If not, then one or multiple new candidate solutions are generated and evaluated again until

one of them reaches a sufficiently good quality. An optimization algorithm is then an algorithm that systematically generates and evaluates candidate solutions.

2.9.1 Optimization problems

In optimization, it is necessary to distinguish between optimization problems and problem instances (Korte and Vygen, 2018). A problem instance describes one particular problem for which an optimal solution needs to be found. In contrast, an optimization problem is a more abstract description of a class of problem instances with similar characteristics. For example, the traveling salesman problem (TSP) is an optimization problem, concerned with finding the shortest routes on a weighted graph (Lin and Kernighan, 1973). A problem instance I is then an instantiation of the optimization problem. In the example of the TSP, a problem instance corresponds to the choice of one particular graph.

Optimization problems can be categorized according to different criteria (Korte and Vygen, 2018). For example, if the objective function and all constraints are linear, the optimization problem is a linear programming problem. In the context of this dissertation, however, it is more important to classify the optimization problems according to the domains of their variables (Korte and Vygen, 2018). If all variables are continuous, then the optimization problem is called a continuous problem. Similarly, if all variables can take only discrete values, then the optimization problem is called a discrete problem. Hybrid problems exist in which part of the variables are discrete and another part is continuous. These problems are called mixed-variable problems. It is to note that optimization problems can sometimes be transformed from one model into another. Therefore, some discrete problems can be modeled as continuous problems and continuous problems can be modeled as discrete ones.

Several problems are commonly modeled as continuous optimization problems. The maximum flow problem is an example of a continuous optimization problem (Edmonds and Karp, 1972). In the maximum flow problem, a directed graph is given. One node is the source, and another node is the sink. All other nodes are intermediate nodes. The edges of the nodes describe the capacities that can flow from one node to the other. The optimization problem is now to assign (real-valued) flows to each edge such that the sink receives the maximum amount of flow, with no node receiving more input than it outputs or outputting more than it receives.

Similarly, many problems can be modeled as discrete optimization problems. They can be further classified into combinatorial problems (in which a candidate solution is represented as a combination of elements, e.g., a permutation of a set) and integer programming problems (in which candidate solutions are represented as vectors of integers). Often, these two classes are related, and a combinatorial

representation can be turned into an integer programming one and vice versa (Korte and Vygen, 2018). Two well-known problems of discrete optimization are the traveling salesperson problem (TSP) and the knapsack problem (Mathews, 1897). In the traveling salesperson problem, an undirected weighted graph is given. The goal of the optimization process is to find the shortest tour that visits all nodes in the graph, given a specific starting node. In the knapsack problem, the goal is to find an optimal combination of items that maximizes their total value while not exceeding the capacity of the sack.

Sometimes optimization problems can be modeled as either continuous or combinatorial problems. Additionally, many real-world applications combine elements from both continuous and discrete optimization. Furthermore, evaluation of candidate solutions might not be deterministic. That is, when evaluating a candidate solution, one cannot assess the actual performance. Rather, one can only observe samples of the random process that defines the quality of a candidate solution. In these cases, the optimization problem is called a *stochastic optimization problem*.

2.9.2 Optimization algorithms

Various algorithms have been developed to optimize classes of optimization problems. These optimization algorithms can be categorized into exact algorithms, heuristic algorithms and metaheuristic algorithms.

Exact methods provably find the optimal solution to an optimization problem, if it exists (Boyd and Vandenberghe, 2004). For example, the simplex algorithm can be used to find optimal solutions to linear programming problems (Dantzig, 1990) and the branch-and-bound algorithm can be used to find optimal solutions in the traveling salesman problem (Padberg and Rinaldi, 1991).

While exact methods are guaranteed to find the optimal solution, there is one major challenge: the computational complexity (Korte and Vygen, 2018). The computation complexity describes in a formal way the (worst-case) run time of an algorithm with respect to the size of the input (i.e., the size of the problem instance). Computational complexity is usually expressed using asymptotic notation. An algorithm has a computational complexity $\mathbb{O}(f(n))$ if its run time grows asymptotically within a constant factor of $f(n)$. In optimization, there are two classes of computational complexity that are of special interest. The complexity $\mathcal{P}$ contains all optimization problems for which there exists an algorithm that finds an optimal solution in polynomial time. The complexity class $\mathcal{NP}$ contains all optimization problems in which a solution c given for an instance I can be verified in polynomial time. It should be clear that $\mathcal{NP} \subseteq \mathcal{P}$, as the construction of a solution in $\mathcal{P}$ could serve as a witness for the verification. However, unless $\mathcal{NP} = \mathcal{P}$, this means that there are algorithms in $\mathcal{NP}$ where a solution cannot be found in polynomial time.

As a result, these problems often cannot be solved exactly in a reasonable time for large problem instances.

Instead of searching for exact solutions that are guaranteed to be optimal, it is often sufficient to find solutions that are close to the optimal one. In these cases, one can rely on the use of heuristic optimization algorithms. Heuristic optimization algorithms do not necessarily find an optimal solution in finite time. Instead, they iteratively generate and evaluate candidate solutions. The generation of new candidate solutions is then guided by heuristic information about the optimization problem. Heuristic algorithms have shown to provide near-optimal solutions often significantly quicker than exact algorithms (Glover and Kochenberger, 2003). A special kind of heuristics are metaheuristic algorithms. They are algorithms that find approximate solutions to optimization problems without explicitly relying on domain knowledge about the optimization problem. Often, an actual implementation of a metaheuristic might rely on heuristic information to improve performance on a specific optimization problem, but metaheuristics are principally applicable to larger classes of optimization problems.

2.9.3 Metaheuristics

Metaheuristic algorithms are general optimization algorithms that are applicable to various classes of optimization problems (Glover and Kochenberger, 2003). That is, the general search strategy is independent of the optimization problem. Nevertheless, some operators might need to be specified specifically for the optimization problem at hand and the performance of the optimization algorithm might depend in parts on the choice of these problem-specific components.

Local search algorithms search the neighborhood of candidates for promising new candidate solutions. The simplest local search algorithm is *iterative improvement*, sometimes also called hill climbing. Iterative improvement maintains an incumbent candidate solution that is the best encountered candidate solution so far. The algorithm systematically explores the neighborhood around the incumbent candidate solution. If it finds a challenger that is strictly better than the incumbent, it accepts the challenger, and the challenger becomes the new incumbent. When there is no further improvement possible in the neighborhood of the incumbent, iterative improvement terminates and returns the incumbent. It is trivial to see that iterative improvement converges to a local optimum. However, it is not guaranteed that the local optimum corresponds to a global one or the quality of the local optimum is close to that of the global one. Several other local-search-based metaheuristics have been developed to avoid convergence to local optima.

In general, it is necessary to find a trade-off between exploration and exploitation. *Exploitation* is the use of knowledge about high-quality regions of the search space, whereas *exploration* describes a behavior that operates independently of

any knowledge about the search space. Iterative improvement is an algorithm that solely relies on exploitation and features no exploration. As a result, it has no means of escaping a local optimum. On the other hand, an algorithm that would solely focus on exploration would be equal to a random search. Most optimization algorithms therefore contain components that try to exploit the knowledge about higher quality regions of the search space while maintaining an element of exploration to escape local optima. Several local-search based metaheuristics have been developed, such as tabu search (Glover, 1989), simulated annealing (Kirkpatrick et al., 1983), iterated local search (Lourenço et al., 2003), greedy randomized adaptive search procedure (Feo and Resende, 1989), and variable neighborhood search (Mladenović and Hansen, 1997).

Besides local searches, metaheuristic optimization algorithms also encompass population-based optimization algorithms. Population-based algorithms address the issue of convergence towards local optima by maintaining a set (or *population*) of candidate solutions in possible different regions of the search space. When one individual candidate solution gets stuck in a local optimum, other candidate solutions can still explore the search space. In order to search more efficiently, the individuals in the population share information between each other in order to bias other individuals towards higher performing regions of the search space. *Evolutionary algorithms* are a commonly used class of population-based algorithms (Bäck et al., 1997). Inspired by the natural process of evolution, evolutionary algorithms guide the search through means of selection, recombination, mutation, and replacement. Selection and replacement are mechanisms that manipulate the population and aim to bias the search towards higher quality regions of the search space by deciding which candidate solutions are selected for reproduction (selection) or which ones are selected for the next generation (replacement). Reproduction (or crossover) and mutation describe operators that generate new candidate solutions. Reproduction combines the information of two or more parents to generate a new offspring (exploitation), whereas mutation randomly changes one candidate solution (exploration). Other population-based metaheuristics include particle swarm optimization (Eberhart and Kennedy, 1995), ant colony optimization (Dorigo et al., 1996; Dorigo and Stützle, 2014), and memetic algorithms (Moscato and Cotta, 2010).

2.10 Automatic algorithm configuration

A recent application of metaheuristics is automatic algorithm configuration (Birattari et al., 2002; Birattari, 2004; Pérez Cáceres, 2017). It is necessitated by the fact that many instances of algorithms have parameters that influence its performance in specific circumstances. Manually finding the optimal set of parameters has

proven infeasible. In automatic algorithm configuration, the problem of finding the optimal set of parameters is solved automatically by using an optimization algorithm. Automatic algorithm configuration is a more general case of the problem in machine learning that is often known as hyperparameter optimization (Feurer and Hutter, 2019). Automatic algorithm configuration problems often have both discrete and continuous domains. Furthermore, it is often the case that the evaluation is stochastic, that is, the quality depends on other (possibly random) factors outside of the parameters.

In the literature, several optimization algorithms have been successfully applied to automatically tune these algorithms for improved performance. For example, Pérez Cáceres et al. (2018) used Iterated F-race (López-Ibáñez et al., 2016) to tune the parameters of the compiler GCC, resulting in compiled machine code whose run time was improved by up to 40% when compared to code compiled with the `-O2` and `-O3` flags. Other optimization algorithms used in automatic algorithm configuration include F-race (Birattari et al., 2002), ParamILS (Hutter et al., 2009), GGA (Ansótegui et al., 2009), and SMAC (Hutter et al., 2011; Lindauer et al., 2022). Hutter et al. (2009) used ParamILS to tune the parameters of CPLEX, resulting in improved run time on several benchmarks. And KhudaBukhsh et al. (2016) tuned the parameters of SAT solvers from components. Similarly, Ansótegui et al. (2009) used ParamILS and GGA to tune several SAT solvers. Thornton et al. (2013) developed Auto-WEKA, an automatic algorithm configuration framework for the WEKA library based on SMAC. For a recent survey of automatic algorithm configurators, see, for example, the work of Schede et al. (2022).

Chapter 3

Methods

In this chapter, I present common elements of the methodology that I applied throughout the experiments presented in this dissertation. The chapter is structured as follows. In Section 3.1, I present the optimization algorithm Iterated F-race. In Section 3.2, I present the version of the e-puck robot I used in my experiments, and in Section 3.3, the reference model that formalizes access to the sensors and actuators. In Section 3.4, I present AutoMoDe-`Chocolate`, an automatic modular design method that forms the basis for the design methods I developed. In Section 3.5, I present `EvoStick`, a yardstick implementation of neuro-evolution. In Section 3.6, I present the experimental environments used in my experiments. In Section 3.7, I describe the statistical analysis performed on the results.

3.1 Iterated F-race

Iterated F-race (Balaprakash et al., 2007; López-Ibáñez et al., 2016) is an optimization algorithm that searches the space of all possible candidate solutions for the best one according to a mission-specific measure of performance. The algorithm executes several iterations, each iteration reminiscent of a race (Maron and Moore, 1997).

In the first race, a uniformly distributed set of candidate solutions is sampled. These candidates are initially evaluated on a set of instances. Typically, an instance describes the configuration of the arena at the beginning of an experiment (that is, positions and orientations of the robots, positions of eventual obstacles or objects of interest, and color of the floor).

After the evaluation is performed, a Friedman test (Friedman, 1937, 1939; Conover, 1999) is performed on the performance obtained by the candidate solutions. The candidate solutions that performed significantly worse than at least another one are discarded. The algorithm keeps evaluating the remaining candidate

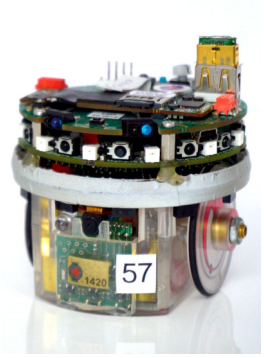


Image credit: David Garzón Ramos.

Figure 3.1: The e-puck robot used in my experiments. It is augmented with a range-and-bearing board and a Linux extension board.

solutions on new instances and discards those that are statistically dominated.

The race terminates when only one surviving candidate solution remains, or when the maximal number of evaluation defined for the race is reached. In the following races, the new set of candidate solutions is sampled with a distribution that gives higher priority to solutions that are similar to the surviving solutions of the previous one.

An implementation of Iterated F-race is provided by the `irace` package[1] (López-Ibáñez et al., 2016), written in R (R Development Core Team, 2008).

3.2 The e-puck robot

All experiments were performed on an extended version of the e-puck robot (see Figure 3.1), a cylindrical, two wheeled robot with a diameter of approximately $0.14\,\mathrm{m}$ (Mondada et al., 2009; Garattoni et al., 2015). The robot is extended with a Linux extension board[2] and a range-and-bearing board (Gutiérrez et al., 2009).

The e-puck has access to a few sensors and actuators. Namely, the base version of the e-puck has eight infra-red transceivers located around the body of the e-puck. They can be used to detect the proximity of other objects as well as perceive ambient light. Other sensors of the e-puck are a microphone, a 3-axis accelorometer, and a front-facing camera. Furthermore, the e-puck was augmented with three ground sensors that can detect the grey-scale color of the floor and a range-and-bearing board (Gutiérrez et al., 2009). The range-and-bearing board uses infrared transmitters and receivers located around the perimeter of the robot

[1] `https://mlopez-ibanez.github.io/irace/`

[2] `https://www.gctronic.com/doc/index.php/Overo_Extension`

Table 3.1: Reference model RM1.1 (Hasselmann et al., 2018), which formalizes the sensors and actuators of the e-puck robot. The period of the control cycle is 100 ms.

Sensor/Actuator	**Parameters**	**Values**
proximity	$prox_i$, with $i \in \{0, \ldots, 7\}$	$[0, 1]$
light	$light_i$, with $i \in \{0, \ldots, 7\}$	$[0, 1]$
ground	$ground_i$, with $i \in \{0, \ldots, 2\}$	$\{black, gray, white\}$
range-and-bearing	n	$\{0, \ldots, 19\}$
	V_d	($[0, 0.5]$ m, $[0, 2\pi]$ radian)
wheels	v_l, v_r	$[-0.12, 0.12]$ m/s

to transmit and receive messages between the robots. Additionally, it provides localization information, such as distance and orientation, of received messages. As for actuators, the e-puck has two wheels, the speeds of which can be set independently of each other. It also has a ring of eight red LEDs distributed around the body of the robot.

The e-puck could be further augmented, for example with an omnidirectional vision turret (École polytechnique fédérale de Lausanne, 2010). In the context of this work, I only used the proximity sensors (for light and proximity readings), the ground sensors, the range-and-bearing board and the wheels. I used the firmware and ARGoS3 plug-in developed at IRIDIA (Garattoni et al., 2015; Ligot et al., 2017).

3.3 Reference model RM1.1

Access to the sensors and actuators of the e-puck was formalized through a reference model, namely RM1.1 (see Table 3.1). Reference models allow programming several design methods against the same interface, providing a unified interface to sensors and actuators. In RM1.1, the robot has access to 8 proximity sensors ($prox_i$) that can perceive obstacles and other robots up to a distance of $0.1\,\mathrm{m}$, 8 light sensors ($light_i$) that can perceive ambient and directional light, 3 ground sensors ($ground_i$) that can detect if the floor is white, black or gray, and a range-and-bearing board that allows the robot to perceive its neighbors up to a range of $0.5\,\mathrm{m}$, in the form of the number of neighbors (n) and a vector towards their collective center of mass (V_d). The reference model also allows setting the velocities of the two wheels independently of each other (v_l, v_r).

3.4 Chocolate

Chocolate is a design method that designs control software for the e-puck robot (see Section 3.2) (Francesca et al., 2015). The capabilities of the robot are abstracted into a reference model (see Section 3.3). Based on the reference model, a set of behavioral and conditional modules is defined (see Section 3.4.1). Chocolate then uses Iterated F-race (see Section 3.1) to automatically assemble these modules into finite-state machines (see Section 3.4.2).

3.4.1 Modules

Chocolate has at its disposal a set of six behavioral modules and six conditional modules (Francesca et al., 2015). A behavioral module performs a low-level behavior in which the robot operates its actuators in response to the readings of its sensors. A conditional module performs a check on the context that the robot perceives via its sensors. Conditions contribute to determine which behavior is executed at any moment in time. If a module has parameters, they are fine-tuned by the optimization algorithm.

In the following, I briefly describe the low-level behaviors and conditions. For the details, I refer the reader to their original description given by Francesca et al. (2014).

Behavioral modules

Exploration: if the front of the robot is clear of obstacles, the robot goes straight. When an obstacle is perceived via the front proximity sensors, the robot turns in-place for a random number of control cycles drawn in $\{0, ..., \tau\}$. τ is an integer parameter $\in \{0, ..., 100\}$.

Stop: the robot does not move.

Phototaxis: the robot goes towards the light source. If no light source is perceived, the robot goes straight while avoiding obstacles.

Anti-Phototaxis: the robot goes away from the light source. If no light source is perceived, the robot goes straight while avoiding obstacles.

Attraction: the robot goes towards its neighboring peers, following αV_d, where $\alpha \in [1, 5]$ controls the speed of convergence towards them. If no peer is perceived, the robot goes straight while avoiding obstacles.

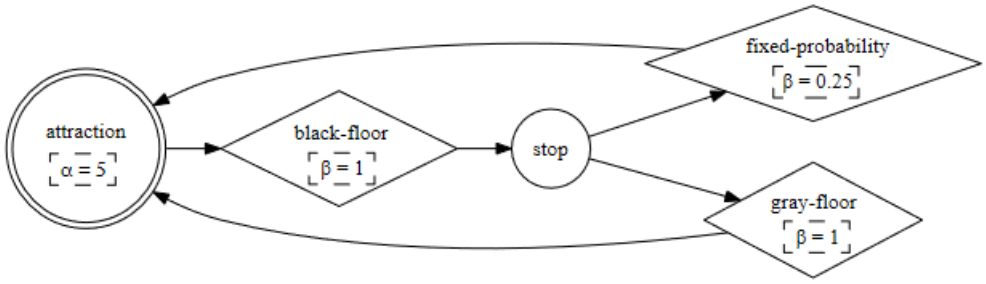


Image originally published by Francesca et al. (2014).

Figure 3.2: An example of a finite-state machine that `Chocolate` can generate. Circles represent states of the finite-state machine and arrows represent transitions. The diamonds represent the conditions associated with the transitions.

Repulsion: the robot goes away from its neighboring peers, following $-\alpha V_d$, where $\alpha \in [1, 5]$ controls the speed of divergence. If no peer is perceived, the robot goes straight while avoiding obstacles.

Conditional modules

Black Floor: true with probability β, if the ground situated below the robot is perceived as black.

Grey Floor: true with probability β, if the ground situated below the robot is perceived as gray.

White Floor: true with probability β, if the ground situated below the robot is perceived as white.

Neighborhood Count: true with probability $z\left(n\right) = \left(1 + e^{\eta(\xi - n)}\right)^{-1}$, where n is number of detected peers. The parameters $\eta \in [0, 20]$ and $\xi \in \{0, ..., 10\}$ control the steepness and the inflection point of the function, respectively.

Inverted Neighborhood Count: true with probability $1 - z\left(n\right)$.

Fixed Probability: true with probability β.

3.4.2 Control architecture

In the context of this work, I will refer to instances of control software as finite-state machines, if they can be formally expressed as probabilistic finite-state machines. Probabilistic finite-state machines are composed of states and transitions. Each

Table 3.2: Noise models for the design and pseudo-reality context.

Sensor/Actuator	Design model	Pseudo-reality model
proximity	0.05	0.05
light	0.05	0.90
ground	0.05	0.05
range-and-bearing	0.85	0.90
wheels	0.05	0.15

state has an associated behavior that is executed as long as the state is active. Transitions connect two states and have an associated condition, which can trigger probabilistically. If the transition triggers, then the current state will become inactive and the state at the other end of the transition will become active and will start to execute its associated behavior. `Chocolate` assembles finite-state machines with up to four states, wherein each state can have up to four outgoing transitions. An example of such a finite-state machine is shown in Figure 3.2. The displayed finite-state machine has two states and a total of three conditions. The associated behavior of the initial state is the Attraction module. Its parameter α is set to 5. The initial state has exactly one outgoing transition, with the Black Floor conditional module, leading to the second state with the Stop module. The second state has two outgoing transitions pointing back to the initial state, one with the Grey Floor condition and the other with the Fixed Probability probability.

3.5 `EvoStick`

`EvoStick` is a yardstick implementation of evolutionary swarm robotics (Francesca et al., 2012, 2014, 2015). The robots are controlled by an artificial neural network with a fixed topology. The artificial neural network contains 24 input and 2 output nodes that are mapped to the possible inputs and outputs as defined by the reference model RM1.1. It is fully connected, feed-forward, and does not contain hidden layers. `EvoStick` uses an evolutionary algorithm to optimize the weights of this artificial neural network.

3.6 Experimental environment

Simulations were performed with ARGoS3, version beta 48 (Pinciroli et al., 2012; Garattoni et al., 2015). ARGoS3 is a light-weight, physics-based simulator for swarm robotics. Following best practices, I used realistic noise settings for the sensors and actuators (see Table 3.2). The noise settings were used in several

prior studies (Francesca et al., 2014, 2015; Hasselmann and Birattari, 2020; Spaey et al., 2020). The noise on the range-and-bearing board was implemented as a probability that a message is not received by a robot. The noise on the other sensors was implemented as an additive noise, sampled from a Gaussian distribution with mean 0 and variance according to the noise setting.

Pseudo-reality is a concept to evaluate the transferability of control software (Ligot and Birattari, 2020, 2022). Instead of assessing the performance directly in reality, a different simulation context is used. Research has shown that control software that transfers well into reality also transfers well into pseudo-reality, while control software that transfers badly into reality also transfers badly into pseudo-reality. The pseudo-reality used in my experiments is an ARGoS3 simulation with the noise values shown in Table 3.2.

Experiments with the real robots were conducted in the ground robot experimental arena at IRIDIA. The ground robot experimental arena is a $70\,\mathrm{m}^2$ space, dedicated to testing and running experiments with ground robots. The arena also contains an overhead tracking system that was used to track the positions of the robots during the experiments (Stranieri et al., 2013; Legarda Herranz et al., 2022). The missions do not take place inside the whole arena. Instead, a bounded environment is placed inside the ground robot experimental arena and the inside of the bounded environment constitutes the arena for each mission.

3.7 Statistical analysis

The results of this dissertation are presented in two forms, giving a visual representation of the samples: notched box-and-whisker boxplots and Friedman plots. In a notched box-and-whisker boxplot, a horizontal line denotes the median of the sample. The lower and upper sides of the box are called upper and lower hinges and represent the first and third quartiles of the observations, respectively. The upper whisker extends either up to the largest observation or up to 1.5 times the difference between upper hinge and median—whichever is smaller. The lower whisker is defined analogously. Small circles represent outliers (if any), that are observations that fall beyond the whiskers. Notches extend to $\pm 1.58 IQR/\sqrt{n}$, where IQR is the interquartile range and n is the number of observations. Notches indicate the 95% confidence interval on the position of the median. If the notches of two boxes do not overlap, the observed difference between the respective medians is significant (Chambers et al., 1983).

Friedman plots show the estimated median rank for each design method, as determined by a Friedman test. Ranks are computed on a per-mission basis. This allows comparing the performance of the design methods across missions, irrespective of the different scale of the results. The Friedman plots show the mean

rank as a point, and the 95% confidence interval as whiskers. If the whiskers of two methods do not overlap, than the difference between their median ranks is statistically significant.

Statements such as “method A outperforms method B” or “method A performs significantly better than method B” are based on a paired two-sided Wilcoxon signed-rank tests with a confidence level of 95%.

Chapter 4

Local-search based optimization algorithms

Despite their simplicity, local search algorithms have shown promising results in many optimization problems. For this reason, I developed two automatic modular design methods, AutoMoDe-`Cherry` (see Section 4.3) and AutoMoDe-`IcePop` (see Section 4.4), based on local search algorithms. Both design methods are based on `Chocolate` and only differ in the optimization algorithm employed. `Cherry` uses iterative improvement (see Section 4.1.1) and `IcePop` uses simulated annealing (see Section 4.1.2). As both design methods are based on local search algorithms, they require a formal definition of the neighborhood (see Section 4.2).

4.1 Local search algorithms

Local search algorithms are a class of optimization algorithms that operate on the following principles (Glover and Kochenberger, 2003). Let P be the optimization problem. P defines a set of solutions, which can take different forms such as variable assignments, ordering of elements, or graph cuts. Often P also defines a set of constraints that restrict the set of solutions. A candidate solution is a solution that satisfies all constraints. Let C denote the set of all candidate solutions. The objective function $f_P : C \rightarrow \mathbb{R}$ of P assigns a quality measure to each candidate solution.

The goal is to find a solution $c \in C$ that is optimal with respect to f_P. This is either a global minimum $(f_P(c) \leq f_P(c'), \forall c' \in C)$ or a global maximum $(f_P(c) \geq f_P(c'), \forall c' \in C)$, depending on the definition of f_P. However, any maximization problem can be transformed into a minimization problem (and vice versa) as $argmax(f_P) = argmin(f'_P)$ for $f'_P(x) = -f_P(x)$. In the following, I will assume that the objective function is to be maximized.

Algorithm 4.1: Iterative Improvement.

input : c_0 - initial candidate solution
output : c_{best} - best encountered candidate solution

```
c_best ← c_0
while not TerminationCriterion do
    c_perturbed ← Exploration(Neighborhood(c_best))
    if Acceptance(c_best, c_perturbed) then
        c_best ← s_perturbed
    end
end
return c_best
```

For every candidate solution c, a neighborhood $N(c)$ is defined. $N(c)$ contains all candidate solutions that can be reached from c through one step of the local search algorithm. Starting from an initial candidate solution, the local search algorithm tries to move from the incumbent candidate solution c to a candidate solution $c' \in N(c)$. The selected candidate solution c' is called the perturbed candidate solution when the neighborhood $N(c)$ is described through the application of perturbation operators. The perturbed candidate solution is accepted—and becomes the new incumbent candidate solution—according to an acceptance criterion. Once a termination criterion is met, the current incumbent candidate solution is returned. The returned solution often is not necessary the global optimum, but only a local optimum ($f_P(c) \geq f_P(c'), \forall c' \in N(c)$), that is there is no better candidate solution in the neighborhood of c, but it could be that a candidate solution outside of the neighborhood of c has a better solution quality.

In my work, I considered two local search algorithms, iterative improvement (see Section 4.1.1) and simulated annealing (see Section 4.1.2).

4.1.1 Iterative improvement

Iterative improvement is a simple local search technique that accepts a perturbed candidate solution if and only if it is better than the current one (see Algorithm 4.1). There still remains some freedom in implementing this template, like the function `Exploration` which defines how the neighborhood is explored. For example, the neighborhood can be explored randomly, exhaustively, or according to some heuristic. The acceptance criterion `Acceptance` defines under which circumstances the perturbed solution is selected as the new best solution. In problems with deterministic performance measures, this might be as simple as comparing the performances of the two solutions. If the performance measure is stochastic,

Algorithm 4.2: Component-based simulated annealing algorithm

input : c_0 - initial candidate solution
output : c_{best} - best encountered candidate solution

```
T_0 ← InitialTemperature
ĉ ← c_0
while not TerminationCriterion do
    c_i ← Exploration(Neighborhood(ĉ))
    if Acceptance(ĉ, c_i, T_i) then
        ĉ ← c_i
        if c_i improves over c_best then
            c_best ← c_i
        end
    end
    if TemperatureLengthReached then
        T_{i+1} ← Cooling(T_i)
    end
    if TemperatureRestartReached then
        T_{i+1} ← TemperatureRestart(T_i)
    end
end
return c_best
```

the acceptance criterion could require comparing the mean or median of a sample of performances, or selecting randomly, instead of deterministically, with regard to the observed performance. The function `TerminationCriterion` remains another choice. It is used to define an end for the potentially time-consuming search for improvements. Depending on the problem context, this could, for example, be a known optimum value, a criterion that models the passage of time (e.g., runtime of the algorithm, budget of perturbations), or a criterion that detects stagnation (e.g., indicating that a local optimum has been reached). Furthermore, the iterative improvement algorithm features two problem-specific elements: c_0, the initial candidate solution, and the function `Neighborhood` that computes the neighborhood for any given candidate solution.

4.1.2 Simulated annealing

Simulated annealing (Kirkpatrick et al., 1983) is a well-studied algorithm (Burke and Bykov, 2017; Hajek, 1988; Lundy and Alistair, 1986; Mitra et al., 1985; Nikolaev and Jacobson, 2010) that has found many applications (Aarts et al., 2005;

Nikolaev and Jacobson, 2010). It is a metaheuristic inspired by the thermodynamical process of annealing. At higher temperatures the particles in a crystal are more excited and can move more freely than at lower temperatures. Similarly, the simulated annealing algorithm has a *temperature* parameter. When the temperature is high, the algorithm has a chance to accept worsening solutions, mimicking the free movement of the particles. At lower temperatures, the algorithm will select worsening solutions less likely, thus constraining the movement of the solution candidate. Simulated annealing has shown properties that are desirable for the automatic design of control software. It has been shown to effectively traverse the search space and to converge quickly towards promising solutions (Hoos and Stützle, 2005). This allows an efficient use of the allocated budget. Furthermore, simulated annealing contains mechanisms to escape local optima—e.g., by accepting worsening moves at higher temperatures. This is an important property as it reduces the risk of premature convergence to suboptimal solutions.

Algorithm 4.2 shows the component-based simulated annealing algorithm, first proposed by Franzin and Stützle (2019) to unify several variants of the simulated annealing algorithm. The component-based simulated annealing algorithm contains placeholders for problem-specific definitions and commonly used components. Similarly to iterative improvement, an initial candidate solution and a `Neighborhood` function must be specified. Like iterative improvement, simulated annealing also requires the definition of a termination criterion (`TerminationCriterion`), an exploration scheme (`Exploration`), and an acceptance criterion (`Acceptance`). The termination criterion and exploration scheme are defined exactly as for iterative improvement. The acceptance criterion, however, does not only depend on the incumbent and the perturbed candidate solution but also on the current temperature. In order to control the behavior of the temperature, simulated annealing introduces some additional components that must be chosen. The initial temperarature (`InitialTemperature`) defines the initital value that will be assigned to the temperature parameter. Periodically, the temperature will be updated, according to the cooling scheme (`Cooling`). The temperature length (`TemperatureLengthReached`) then describes the frequency with which the temperature will be updated. Similarly, the temperatue can be restarted (`TemperatureRestart`) periodically, increasing it again.

4.2 Neighborhood structure

Local search algorithms search through the neighborhood of a candidate solution for a better one. I defined the neighborhood implicitly through a set of perturbation operators. For a given candidate solution (that is, an instance of control software), the neighborhood function can be derived through a systematic and exhaustive

application of the perturbation operators.

The operators are presented in a deterministic fashion and with fixed parameters. Wherever there is a choice, it will either be presented as a fixed parameter or ignored, if not directly relevant to the description of the operator. In any implementation, a choice mechanism must be decided upon. In this work, the choices will be made through random and independent sampling from a uniform distribution over all eligible values.

The perturbation operators follow a hierarchy (structural $>$ modular $>$ parametric) where no lower level should change any properties of a higher one. On the lowest level are parametric perturbations. These perturbations change only the parametric values of the modules (within the limits defined by the design method), that are associated with the nodes of the instance of control software. They are not allowed to change the choice of modules themselves or alter the structure of the control software. The middle level contains modular perturbations. These perturbations change the modules that are associated with the nodes, but without changing the transitions graph between the modules. They can, however, influence the parametric values of the modules—e.g., when replacing one behavior with another, as not all behaviors have the same parameter space. The highest level are structural perturbations that change the graph representation of the control software (either the state transition graph defined by the finite-state machine or the tree representation of the behavior trees). These perturbations necessarily influence also both the modular and parametric levels of the control software.

Validity

In the context of this work, a finite-state machine is considered a valid instance of control software if it fulfills the following criteria:

V1) Correct number of states
   a) **Minimum number of states**: The finite-state machine has at least 1 state.
   b) **Maximum number of states**: The finite-state machine has at most 4 states.
   c) **Initial state**: Exactly one state of the finite-state machine is designated as the initial state.

V2) Correct configuration of states
   a) **Unique behavior**: Each state has exactly one associated behavior from the set of modules.
   b) **Correct parameters**: The parameters of the associated behavior of each state are within the bounds defined for that behavior.

V3) Correct number of transitions
   a) **Minimum number of outgoing transitions**:

1) If there are at least two states in the finite-state machine, then there is no state that has no outgoing transition.
2) If there is exactly one state in the finite-state machine, then this state has no outgoing transition.

b) **Maximum number of outgoing transitions**: Each state has at most 4 outgoing transitions.

c) **Total number of transitions**: There is no limit on the total number of transitions in the graph, other than the implicitly defined $4 \times \#states$.

V4) Correct configuration of the transitions

a) **Unique condition**: Each transition has exactly one associated condition from the set of modules.

b) **Correct parameters**: The parameters of the associated condition of each transition are within the bounds defined for that condition.

c) **Unique starting point**: Each transition has exactly one starting point.

d) **Unique endpoint**: Each transition has exactly one endpoint.

e) **Not self-referencing**: For each transition the starting and endpoint are different.

Perturbation operators

P1) **Add a new transition to the finite-state machine**: Let s be a state, with less than the maximum outgoing transitions and s' be a different state.
Add a transition from s to s.

P2) **Remove a transition from the finite-state machine**: Let t be a transition from s to s', such that t is neither the only outgoing transition from s nor the only incoming transition into s'.
Remove the transition t.

P3) **Add a new state to the finite-state machine**: If the finite-state machine has less than the maximum number of states, let s' and s'' be states, where s' has less than the maximum number of outgoing transitions, and s'' might be the same state as s'.
Add a new state s and add transitions from s' to s and from s to s''.

P4) **Remove a state from the finite-state machine**: Let s be a state that is not the initial state and not an articulation vertex [1] of the state transition graph.
Remove the state s and any incoming and outgoing transitions of s.

P5) **Move the start point of a transition**: Let t be a transition starting in a state s, such that s has at least one other transition t' also starting from it. Let s' be a different state than s that has less than the maximum number of outgoing

[1] In this setting, an articulation vertex is any vertex of the undirected state transition graph, that, if removed, would increase the number of connected components.

transitions.
Change the start point of the transition t from s to s'.

P6) **Move the endpoint of a transition**: Let t be a transition ending in a state s, such that s has at least one other incoming transition t'. Let s' be a different state than s.
Change the endpoint of the transition t from s to s'.

P7) **Change the initial state**: Let s be the initial state and s' be a different state.
Change the current initial state from s to s'.

P8) **Change condition of a transition**: Let t be a transition.
Change the associated condition of t to a different one from the set of modules.

P9) **Change behavior of a state**: Let s be a state.
Change the associated behavior of s to a different one from the set of modules.

P10) **Change parameter of a condition**: Let t be a transition and p a parameter of the associated condition of t.
Set a new value for p within the bounds defined for the condition.

P11) **Change parameter of a behavior**: Let s be a state and p a parameter of the associated behavior of s.
Set a new value for p within the bounds defined for the behavior.

The perturbation operators P1 - P7 are structural perturbations, as they change the state transition graph, defined by the finite-state machine. The perturbation operators P8 and P9 are modular perturbations as they change the behaviors or conditions associated with the states and transitions of the finite-state machine. The perturbation operators P10 and P11 are parametric perturbations as they only affect the parameters of a single module in the finite-state machine.

4.3 AutoMoDe-Cherry

I defined Cherry[2], an automatic modular design method that is based on Chocolate (Francesca et al., 2015). It designs control software for the e-puck robot, formalized through reference model RM1.1 (see Section 3.3). As Cherry is based on Chocolate, it has access to the same set of six modules and six conditions available (see Section 3.4). The six behavioral modules are Stop, Exploration, Phototaxis, Anti-Phototaxis, Attraction, Repulsion. The six conditional modules are Fixed Probability, Neighborhood Count, Inverted Neighborhood Count, Grey Floor, Black Floor, White Floor. These modules are assembled into probabilistic

[2]In the original publication (Kuckling et al., 2020a), no name was chosen to refer to this design method. Here, I retroactively introduce the name Cherry to refer to this design methods and its variants.

finite-state machines with up to four states and up to four outgoing transitions per state.

`Cherry` uses the iterative improvement algorithm (see Section 4.1.1) as optimization algorithm. In the context of `Cherry`, candidate solutions are instances of control software. For the initial candidate solution, I chose a "minimal" behavior. The minimal behavior comprises a finite-state machine with a single state which executes the Stop module. The `Exploration` function that returns the next perturbed candidate solution is defined to apply a random perturbation operator to the incumbent candidate solution. The `Acceptance` function evaluates the incumbent and perturbed candidate solutions ten times on different instances of the mission (initial positions and headings of the robots, defined through a random seed) and accepts the challenger if the mean performance improved. To avoid overfitting to these exact ten seeds, every time a perturbed controller was evaluated, the oldest two seeds were discarded and replaced by two new ones. The perturbed controller is then evaluated on all ten seeds, while the current best controller is evaluated only on the two new seeds, as it has been already evaluated on the old seeds. The perturbed candidate solution then becomes the new incumbent candidate solution and thus the base for the next perturbation. Otherwise, the result of the perturbation is discarded and a new random perturbation is applied to the candidate solution. This process is repeated until a termination criterion is met. For the function `TerminationCriterion`. I chose a computational budget in the form of a maximum number of simulations as termination criterion. That is, the optimization algorithm will terminate when the allocated budget of simulations is exhausted or does not contain enough simulations to perform another comparison. These choices lead to a stochastic hill-climbing search iteratively approaching the first optimum it can find, though not necessarily a global one.

It is possible that the choice of the initial candidate solution can have a significant impact on the performance of iterative improvement. If there exists a local optimum close to the initial candidate solution, it is possible that the optimization algorithm quickly converges towards the local optimum and cannot escape it anymore. Thus, I developed two variants of `Cherry`, `Cherry-Random` and `Cherry-Hybrid`, that differ in their choice of the initial candidate solution. In order to avoid confusion, `Cherry` will also be clarified as `Cherry-Minimal` in the following. `Cherry-Random` uses a randomly generated valid instance of control software as the initial solution. `Cherry-Hybrid` is a hybridization of `Chocolate` and the local search approach already defined for `Cherry-Minimal` and `Cherry-Random`. It operates in two stages, each stage running for one half of the allocated total budget. In the first stage, the hybrid algorithm creates a candidate solution following the protocol described for `Chocolate`—see Section 3.4. The resulting instance of control software is then supplied to the iterative improvement algorithm under the same setup as for `Cherry-Minimal` or `Cherry-Random`

and is then improved for the remaining budget.

Implementations of all design methods are available for download from the supplementary materials page (Kuckling, 2023).

4.3.1 Experiments

I validated the proposed design methods on four missions: AGGREGATION WITH AMBIENT CUES (AAC), SHELTER WITH CONSTRAINT ACCESS (SCA), FORAGING, and GUIDED SHELTER. In order to better appraise the performance of `Cherry-Minimal`, `Cherry-Random`, and `Cherry-Hybrid`, I also used `Chocolate` (see Section 3.4) and `EvoStick` (see Section 3.5) to design control software for the same four missions. All missions were conducted in a dodecagonal arena with walls of $0.66\,\mathrm{m}$ length, enclosing an area of $4.91\,\mathrm{m}^2$. At the beginning of each experimental run, the robots were distributed randomly in this arena and the duration of each experimental run was $120\,\mathrm{s}$.

AAC

In AGGREGATION WITH AMBIENT CUES (AAC), the robots had to aggregate on a black spot. They had two other ambient cues available to help them find the black spot: a light source next to the black spot and a white spot (see Figure 4.1).

The black target region was a circular area with a radius of $0.3\,\mathrm{m}$. It was located in the front half of the arena, $0.6\,\mathrm{m}$ away from the center of the arena. A light source was placed in front of the black spot (outside of the arena). The arena also contained a white circular area with a radius of $0.3\,\mathrm{m}$. It was located in the back half of the arena, $0.6\,\mathrm{m}$ away from the center of the arena.

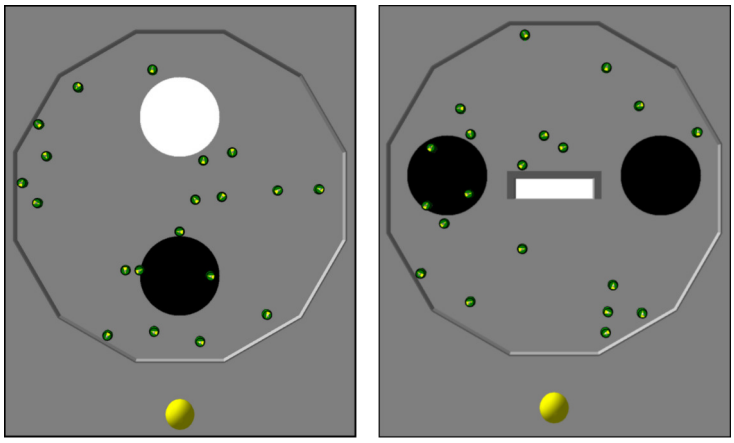
The objective function was defined as:

$$F_{AAC} = \sum_{t=0}^{T} N_t, \tag{4.1}$$

where N_t was the number of robots on the black circle at time step t. It computes the cumulative time that robots spend on the black spot. In order to maximize the objective function, the robots had to aggregate as quickly as possible on the black spot.

SCA

In SHELTER WITH CONSTRAINT ACCESS (SCA) the robots had to aggregate inside a shelter. The arena contained the shelter, a light source, and two black spots (see Figure 4.1).



A) AAC B) SCA

C) Foraging D) Guided Shelter

Image originally published in Kuckling et al. (2020a).

Figure 4.1: Arena layouts for the four missions. The images show ARGoS3 simulations with 20 e-puck robots inside a dodecagonal arena with different floor colors. A light source is located at the lower end of each image.

The shelter was a white rectangular area, of $0.15\,\mathrm{m} \times 0.6\,\mathrm{m}$, bordered by walls on three sides, and only open on one side. The shelter was positioned in the middle of the arena and was open towards the front of the arena. In the front, outside of the arena, was a light source. Two black circular areas with a radius of $0.3\,\mathrm{m}$ were located to the left and right side of the shelter, $0.35\,\mathrm{m}$ away from the edge of the shelter.

The objective function was defined as:

$$F_{SCA} = \sum_{t=0}^{T} N_t. \tag{4.2}$$

where N_t was the number of robots in the shelter at time step t. It computes the cumulative time that robots spend in the shelter. In order to maximize the objective function, the robots, therefore, had to move into the shelter as quickly as possible.

FORAGING

FORAGING represents an abstracted foraging task. In this task, the robots had to transport items from two food sources to the nest (see Figure 4.1). Since an e-puck does not have the capabilities to physically pick up or deposit items, these actions were abstracted to happen when a robot passes over the corresponding area.

A white region in the front side of the arena represented the nest. It covered the whole width of the arena and has a depth of $0.63\,\mathrm{m}$. A light source was located in front of the white region (outside of the arena). Two black circles, with a radius of $0.15\,\mathrm{m}$, represented the food sources. They were positioned $0.45\,\mathrm{m}$ away from the nest area and with a distance of $1.2\,\mathrm{m}$ between them.

The objective function was defined as:

$$F_{For} = N, \tag{4.3}$$

where N is the number of retrieved items. It computes the number of items retrieved by the swarm. In order to maximize the objective function, the robots had to move back and forth between the nest and a food source as many times as possible.

GUIDED SHELTER

In GUIDED SHELTER, the robots had to move into a shelter. They had two ambient cues to guide them towards the shelter: a light source and a conic region providing a path to the shelter (see Figure 4.1).

The shelter was a black rectangular area of size $0.4\,\mathrm{m} \times 0.6\,\mathrm{m}$. The shelter was located in the front part of the arena $0.3\,\mathrm{m}$ away from the front wall of the arena. It

was enclosed with walls on three sides and only open towards the far end of the arena. In front of the shelter (and outside of the arena) was a light source. Behind the shelter was a white conic area, that was defined in such a way that the edges of the conic area go through the front corners of the shelter and meet in the center of the light source.

The objective function was defined as:

$$F_{GS} = \sum_{t=0}^{T} N_t, \tag{4.4}$$

where N_t was the number of robots in the shelter at time step t. It computes the cumulative time that robots spend within the shelter. In order to maximize the objective function, the robots had to move into the shelter as quickly as possible.

Protocol

All design methods were used to automatically produce control software for a swarm of 20 e-puck robots. As the design process is stochastic, I repeated it 10 times for every design method, leading to 10 instances of control software per design method. Each of these 10 instances of control software was then assessed 10 times in the design context and 10 times in a pseudo-reality context to obtain the performance results. Simulations were performed using the simulatior and noise settings described in Section 3.6.

All design methods were tested with design budgets of $12\,500$ (12.5k), $25\,000$ (25k), and $50\,000$ (50k) simulation runs. That is, after exhausting the allocated simulations the design method must have returned their final instance of control software. All results will be represented by notched boxplots (see Section 3.7).

The source code for the experiments, the generated instances of control software, and the details of the performance, such as raw values and statistical tests, are available online as supplementary material (Kuckling, 2023).

4.3.2 Results

Results 12.5k

Figure 4.2 shows the performance of all design methods when they were allocated a budget of $12\,500$ (12.5k) simulation runs.

The performance I observed for `Chocolate`, `Cherry-Minimal`, `Cherry-Random`, and `Cherry-Hybrid` was similar, with few exceptions. In the mission AAC, `Cherry-Hybrid` outperformed `Chocolate` and `Cherry-Minimal`. In the missions FORAGING and SCA, `Chocolate` is outperformed by `Cherry-Minimal` and `Cherry-Random`.

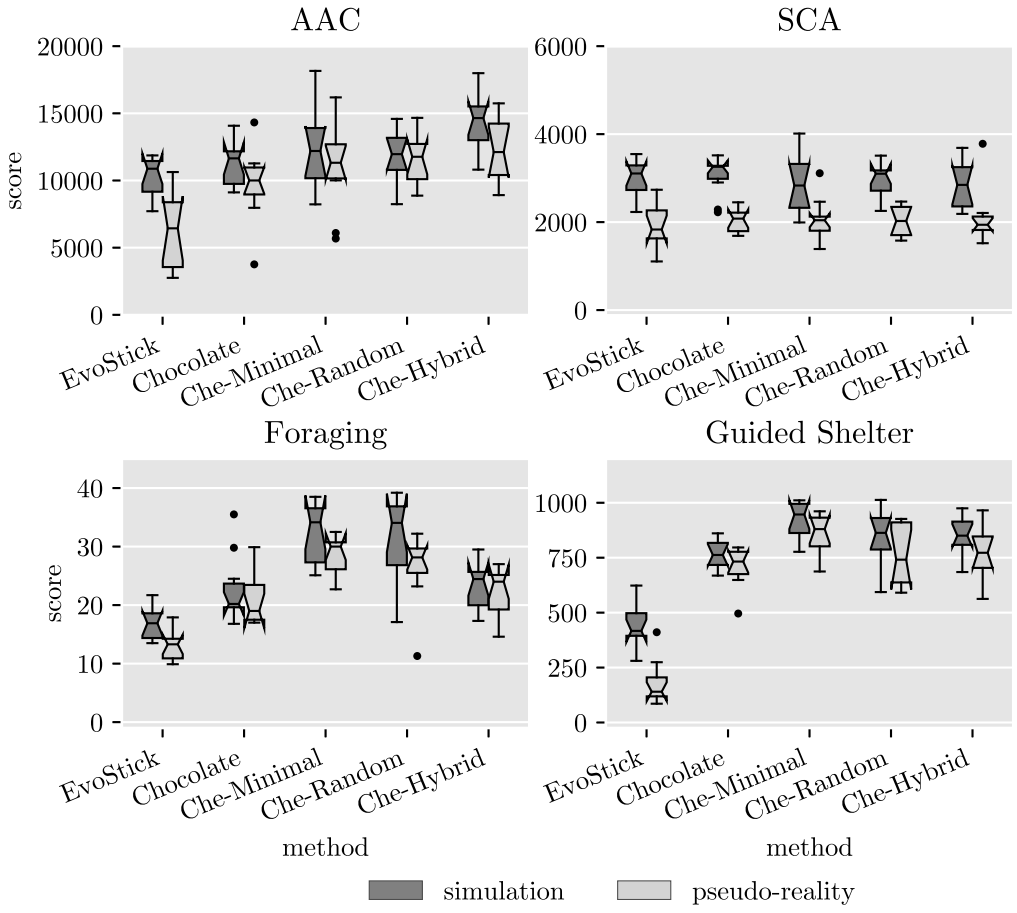


Figure adapted from Kuckling et al. (2020a).

Figure 4.2: Results of all design methods for a budget of 12.5k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

Some of these differences in performance can be explained when looking at the generated control software. For example, in the mission GUIDED SHELTER, where Chocolate was outperformed by Cherry-Minimal and Cherry-Random. Both Cherry-Minimal and Cherry-Random generated finite-state machines that operated on the following principle: A combination of Exploration and Anti-Phototaxis steered the robots onto the white area. Once the robot was on the white area, it would use Phototaxis to move towards the shelter. If the robot left the white area and entered the grey area again, it would fall back to the combination of Exploration and Anti-Phototaxis. The details of this strategy varied from instance to instance, for example, the best instance of control software generated by Cherry-Minimal—see the supplementary material (Kuckling, 2023)—started with the Phototaxis behavior. For Chocolate, on the other hand, the instances of control software only made use of either Anti-Phototaxis or Exploration but never both. Visual inspection of the resulting behavior showed that while instances of control software generated by Chocolate still solved the task sufficiently well, the robots joined the white area on average further away from the shelter than instances of control software that made use of a combination of Anti-Phototaxis and Exploration. Thus the robots took longer to join the nest and therefore achieved lower scores on the objective function.

In all four missions, EvoStick exhibited a large drop in performance when assessed in pseudo-reality. The other design methods did not exhibit such a large drop, except in the mission SCA, where every design method suffered from a large performance drop when assessed in pseudo-reality. This is an indicator that the modular design methods have a good transferability, allowing them to cross the reality gap satisfactorily.

Results 25k

Figure 4.3 shows the results of all design runs with a budget of $25\,000$ (25k) simulations.

In missions AAC, FORAGING, and GUIDED SHELTER, the design methods Cherry-Minimal and Cherry-Hybrid outperformed Chocolate. Additionally, Cherry-Random outperformed Chocolate in the missions FORAGING and GUIDED SHELTER. EvoStick was able to generate sufficiently good control software in the design context, and outperformed several other methods.

Comparison of the generated finite-state machines shows that Chocolate failed to make use of many strategies discovered by the iterative improvement based design methods. In the mission FORAGING, Cherry-Minimal, Cherry-Random, and Cherry-Hybrid made use of Anti-Phototaxis to navigate out of the nest, after dropping off an item. This strategy provided a two-fold benefit. First, it allowed the robots to escape the nest area in less time than if only

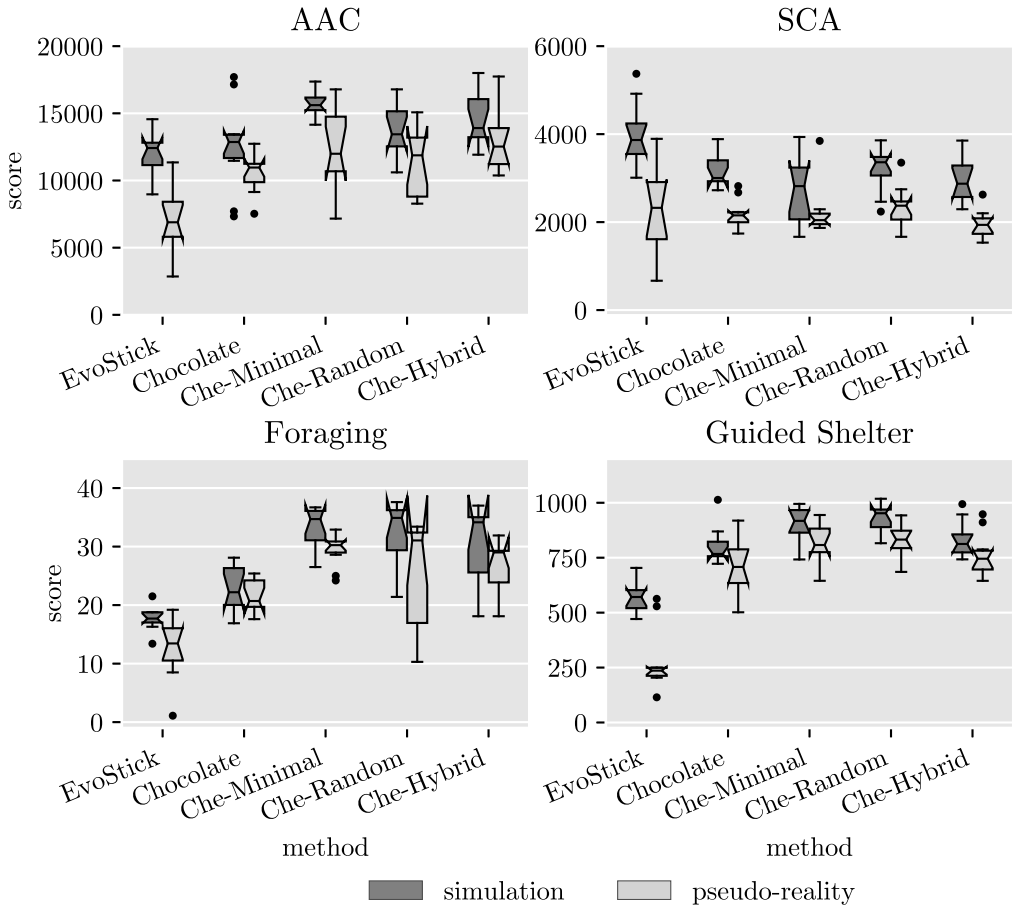


Figure adapted from Kuckling et al. (2020a).

Figure 4.3: Results of all design methods for a budget of 25k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

Exploration was used. In the latter case, the robots would need to encounter first and obstacle (either the wall of the arena or another robot), before they could turn and face towards the rest of the arena again. Secondly, as the robots had been using Phototaxis to navigate from the sources to the nest, Anti-Phototaxis turned the robots in such a way that they are facing towards the source again. While the embedded obstacle avoidance and interference from other robots would reduce the chance of directly going back to the source, this exploitation of the direction seemed to provide a benefit to the performance. Similarly, `Chocolate` also failed to discover the aforementioned Anti-Phototaxis/Exploration strategy in the mission GUIDED SHELTER. In the mission AAC, `Chocolate` employed the same strategies as the other design methods, although it seemed to fail at selecting the appropriate parameters.

When assessed in a pseudo-reality context, however, `EvoStick` was the worst performing design method, being outperformed by all other methods in the three missions AAC, FORAGING, and GUIDED SHELTER. In SCA, `EvoStick` performed best in the design context and although it suffers from a significant drop of performance when assessed in pseudo-reality, it still ranked as one of the best performing design methods in pseudo-reality.

In the mission SCA, the designed finite-state machines suffered from a larger performance drop when assessed in pseudo-reality. This could be an indicator that these instances of control software are overdesigned for the specific design context and would not transfer well into reality.

Results 50k

Figure 4.4 shows the results of all design methods for a budget of $50\,000$ (50k) simulations. All three local search-based design methods, `Cherry-Minimal`, `Cherry-Random`, and `Cherry-Hybrid`, performed similarly through all four missions. Finite-state machines designed by the design methods based on iterative improvement outperformed the finite-state machines that were designed by `Chocolate`, in the missions AAC, FORAGING, and GUIDED SHELTER.

As for the design budget of $25\,000$ simulation runs, `Chocolate` failed to discover some of the strategies employed by the other design methods. Most notably, `Chocolate` neither exploited the Anti-Phototaxis module in FORAGING nor was able to discover the Anti-Phototaxis/Exploration strategy to discover the white area in GUIDED SHELTER more quickly.

Throughout all four missions, `EvoStick` suffered from the largest pseudo-reality gap. The finite-state machines designed by `Chocolate` also experienced small drops of performance, while the finite-state machines generated by iterative improvement showed larger drops of performance when assessed in pseudo-reality. This could be an indicator of potential overdesign for these design methods, how-

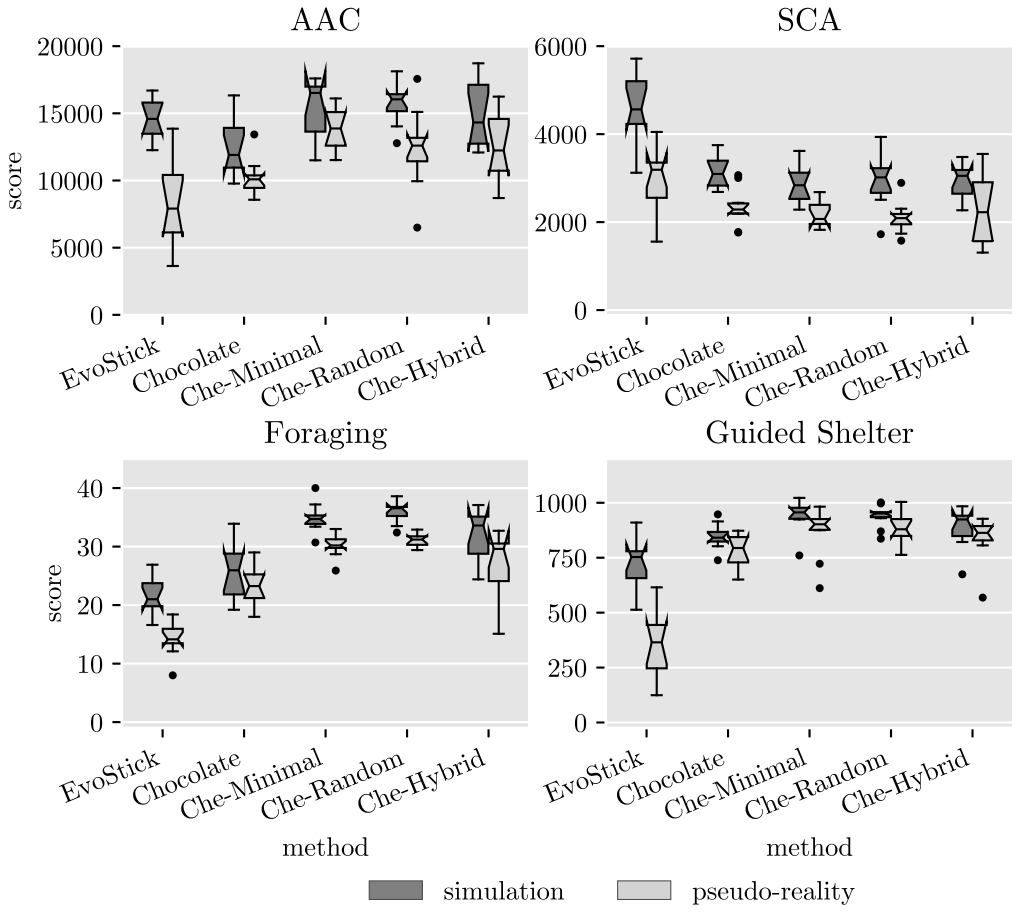


Figure adapted from Kuckling et al. (2020a).

Figure 4.4: Results of all design methods for a budget of 50k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

ever, the effect was not as large as for `EvoStick`.

4.3.3 Discussion

I did not conduct a landscape analysis but the results found by iterative improvement are close to methods known to avoid local optima. This corroborates my hypothesis, that the search space is regular and well-behaved, as I could not find any evidence that iterative improvement remains trapped in local optima, that perform significantly worse than the best know solutions.

However, I did not analyze how thorough the design methods explore this search space. This is mainly due to several limitations that prevent a straightforward analysis. Firstly, it is difficult to quantify how well the search space is explored. For finite-state machines, it is possible that a finite-state machine can be represented through multiple orderings of the states. As no considered optimization algorithm accounts for this kind of equivalency, this leads to the argument that each of these equal representations would need to be counted independently. Yet, discovering one of these representations yields exactly the same performance as discovering any other or even all of the equal representations. This leads to the argument that equivalent representations should only account for a single unit of the search space. Secondly, measuring the similarity between two instances of control software could influence the exploration metric of the search space. Two instances of control software that differ only minimally in exactly one parameter of a condition will most likely show very similar performance, although they are different instances of control software. Again, deciding whether or not these kinds of similarities should be accounted for when determining how well the search space is explored, is a non-trivial decision. Lastly, even if I decided on a method for accounting for similar and equivalent instances of control software in the search space, a simple analysis of the exploration of the search space is meaningless without an analysis of the density and location of sufficiently good solutions in the search space. In fact, many regions of the search space perform poorly with respect to any individual mission. Indeed, even the well-performing instances of control software of one mission are expected to perform poorly when applied to another mission. Therefore, a method exploring 10% of the search space, albeit in the low performing regions, will be less desirable than a method exploring only 1% of the search space, but in the regions that contain the well-performing instances of control software.

For the design methods operating on finite-state machines, `Chocolate` was outperformed by the design methods based on iterative improvement for AAC, FORAGING, and GUIDED SHELTER for budgets of 25k and 50k. A reason for this difference in performance may be that the missions create a search space for the iterative improvement that was easily exploitable by the algorithm. That is, there

seems to be only one way to successfully perform in the missions and the objective function does not contain local optima. This can be seen in the comparison of the two shelter mission SCA and GUIDED SHELTER. In SCA, the robots can only make use of a few indirect cues to determine the position of the shelter. Namely, two spots at the side of the shelter and a light in front of it. The main idea behind these cues is to allow the robots to determine the area in front of and behind the shelter. Yet, a successful shelter finding cannot solely rely on the ambient cues, as it is possible to miss the shelter when doing anti-phototaxis or to miss the black spots when passing from the back of the arena to the front (or starting in the front from the beginning). The mission GUIDED SHELTER, on the other hand, provides a more reliable way to reach the shelter. The white corridor leading to the shelter can be followed more or less exactly using the Anti-Phototaxis behavior and even more so as the robot can detect if it leaves the corridor via the transition to grey floor, allowing the design process to create a contingency behavior for this case. This difference in the difficulty of the task shows also in the performance of the design methods using iterative improvement. In the mission GUIDED SHELTER, the design methods were able to generate well-performing solutions for a budget of only $12\,500$ simulations and higher budgets did not lead to any noticeable improvement. In the mission SCA, on the other hand, all design methods (except `EvoStick`) showed no improvement with higher budgets either. Visual inspection of the generated behaviors showed, however, that the designed control software failed to generated appropriate swarm level behaviors. While some robots ended up in the shelter, the majority of them were distributed throughout the arena. This might have been caused by the reduced amount of cues and additionally by the fact that the shelter could only house a fraction of the swarm.

Across all missions, all design methods based on iterative improvement performed similarly. Differences that were observed for a given mission or a given budget did not generalize to all observed results. This indicates that, for the considered budgets and missions, the starting solution did not seem to play an important role in the solution quality.

4.4 AutoMoDe-IcePop

Together with my co-authors (Kuckling et al., 2020b), I developed AutoMoDe-`IcePop`, an automatic modular design method based on `Chocolate` that uses simulated annealing as optimization algorithm. `IcePop` designs control software for the same robotic platform, uses the same set of modules, and assembles them into probabilistic finite-state machines with the same constraints. The difference between the two methods is that `IcePop` adopts the component-based simulated annealing algorithm (see Section 4.1.2) as the optimization algorithm.

Table 4.1: Configuration of the simulated annealing algorithm.

Component	**Type**	**Parameter**
Initial solution	Minimal controller	Stop behavior
Neighborhood	Defined through perturbation operators	
Initial temperature	Fixed value	$T_0 = 125.0$
Stopping criterion	Budget of simulations	50 000 simulations
Exploration criterion	Random exploration	Valid perturbation operators
Acceptance criterion	Metropolis condition	Mean with 10 samples
Temperature length	Fixed value	$T_{length} = 1$
Cooling scheme	Geometric cooling	$\alpha = 0.9782$
Temperature restart	Fixed value	Every 5000 simulations

Table 4.1 summarizes the choices of components that were used in the implementation of the simulated annealing for `IcePop`. The initial solution supplied to the algorithm was a "minimal" instance of control software. The minimal instance of control software was a finite-state machine with exactly one state executing the Stop behavior. The neighborhood function was implicitly defined through the application of valid perturbation operators (see Section 4.2). The initial temperature was set to 125.0. The stopping criterion was defined as a maximum budget of simulation runs. That is, after the allocated budget of simulation runs was exhausted, the algorithm must return the final instance of control software. The exploration criterion selected a random valid perturbation operator and applied it on the incumbent solution. The acceptance criterion was the Metropolis condition (Kirkpatrick et al., 1983; Metropolis et al., 1953) that accepted or rejected new solutions based on their performance. The Metropolis condition would always accept an improving solution, and would accept a worsening solution with probability $\exp(-(e-e')/T)$, where T is the current temperature, e is quality of the currently best solution and e' is the quality of the perturbed solution. Because the performance of each instance of control software is stochastic, e and e' were computed as the mean of a sample of 10 runs of the respective instance of control software. The temperature length determined the number of steps before the temperature cools down again. The value was set to 1, so that the cooling happened in every step. The cooling scheme, that was then applied, was the geometric cooling (Kirkpatrick et al., 1983). In geometric cooling, the updated temperature is computed as $T * \alpha$, where T is the current temperature and α is the cooling coefficient, which was set as $\alpha = 0.9782$. Additionally, the temperature reset to the initial value every 5000 simulations.

The source code of the implementation of `IcePop` is available as supplementary material (Kuckling, 2023).

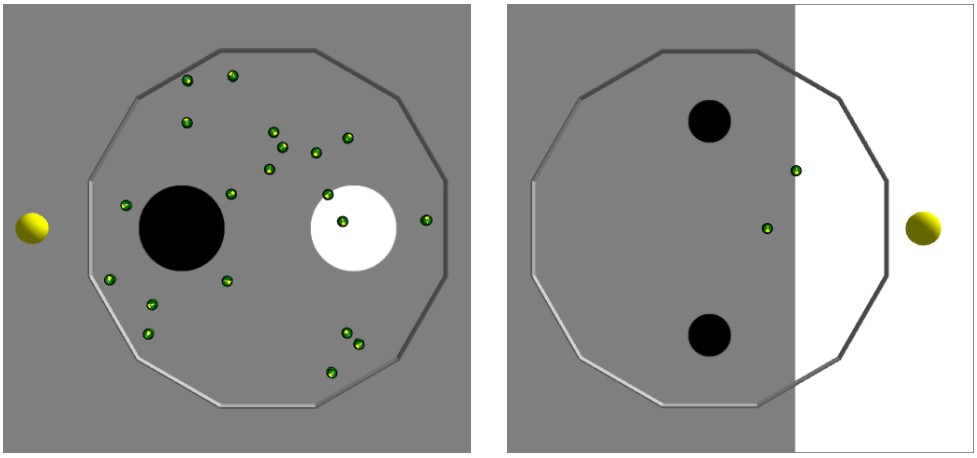


Figure 4.5: The two missions: AAC (*left*) and FORAGING (*right*).

4.4.1 Experiments

`IcePop` and `Chocolate` were used to produce control software for two missions: AGGREGATION WITH AMBIENT CUES (AAC) and FORAGING. All missions were conducted in a dodecagonal arena with walls of $0.66\,\mathrm{m}$ length, enclosing an area of $4.91\,\mathrm{m}^2$. At the beginning of each experimental run, the 20 e-puck robots were distributed randomly in this arena and the duration of each experimental run was $250\,\mathrm{s}$.

AAC

The arena contained two circles, one black, one white. A light source was placed on the side of the arena that contains the black circle (Figure 4.5, left). The robots were tasked to aggregate on the black spot. The objective function was defined as

$$F_{\mathrm{AAC}} = \sum_{t=0}^{T} N_t, \tag{4.5}$$

where N_t was the number of robots on the black circle at time step t.

FORAGING

The arena contained two source areas in the form of black circles and a nest, as a white area. A light source was placed behind the nest to help the robots to navigate (Figure 4.5, right). As the robots have no gripping capabilities, I considered an idealized version of foraging, where a robot was deemed to retrieve an object when

it enters a source and then the nest. The goal of the swarm was to retrieve as many objects as possible. The objective function was defined as:

$$F_{For} = N, \tag{4.6}$$

where N is the number of retrieved items.

Protocol

Four studies were performed, to assess the performance of `IcePop`. The first three studies were designed to analyze the influence of different parameters on the performance of `IcePop`. The fourth study aimed to compare the performance of `IcePop` to `Chocolate`.

In the first study, the influence of the budget on the performance of the generated control software was investigated. Designs with a smaller budget usually need less time to finish but often produce results that perform less well in simulation. The higher the time the better usually the performance in simulation, but an overdesigning effect might be observed, where the improvement in simulation does not carry over to reality. Therefore, `IcePop` was tasked to design control software with five different budgets (5000, $10\,000$, $25\,000$, $50\,000$ and $100\,000$ simulations respectively).

In a second experiment, the influence of the sample size on the performance of the generated control software was investigated. Smaller sample sizes use less of the budget to evaluate one solution, allowing more solution candidates to be investigated. On the other hand, outliers will have a greater impact on the mean of the samples and thus the perceived performance. Larger sample sizes lead to the inverse effect. Fewer total solution candidates would be investigated but the performance of each individual solution candidate is more robust to outliers. The influence of the sample size on the performance of the generated control software was investigated by running designs for three sample sizes: 5, 10, and 15. Additionally all three variants were run for three budgets, namely $25\,000$, $50\,000$, and $100\,000$ simulations.

In a third experiment, the influence of the restarting mechanism was investigated. Restarting resets the temperature to a higher value, allowing the design process to make larger movements in the search space again. Four different restarting mechanisms were considered: fixed length (restarts after a fixed number of simulations, in this case every 5000 simulations), no restart (the temperature cools over the whole design process and is never restarted), reheat (the temperature is reset every 5000 simulations, the new temperature is set to the one that generated the biggest improvement so far), restart once (after the half of the budget is exhausted the temperature resets). All restarting mechanisms were tested for budgets of $25\,000$, $50\,000$, and $10\,000$ simulations.

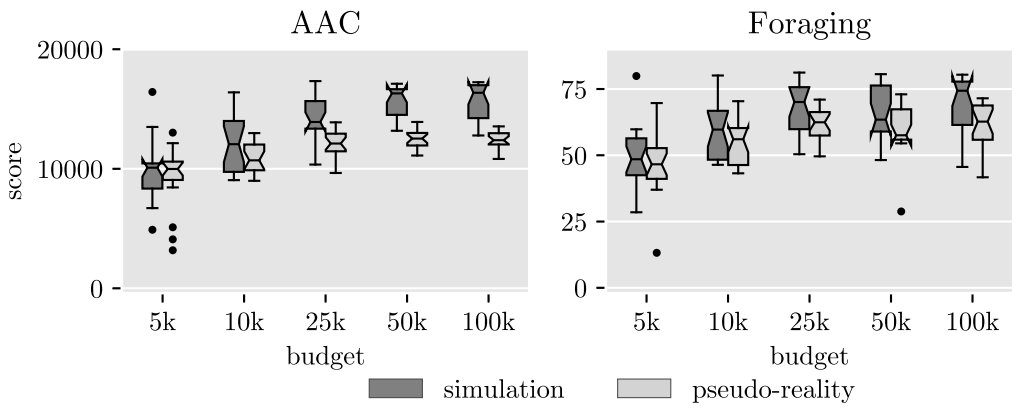


Figure adapted from Kuckling et al. (2020b).

Figure 4.6: Performance of control software created by `IcePop` for different budgets. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

In the last experiment, the performance of `IcePop` was compared with the one of `Chocolate` across three different budgets (25 000, 50 000 and 100 000 simulations).

As each design process is stochastic, 20 independent designs for each design method were performed, resulting in 20 instances of control software. The obtained instances were then each assessed 10 times in the design context (*simulation*) and another 10 times in a pseudo-reality setting (see Section 3.6).

The instances of control software produced, the details of their performances, and videos of their execution on the robots are available as online supplementary material (Kuckling, 2023).

4.4.2 Results

Influence of the budget

The results displayed in Figure 4.6 show the influence of the budget on the performance of the control software generated by `IcePop`. One trend that is apparent from the data, is that, as expected, a larger design budget led to control software that performed better in simulation. However the relative improvement diminished and the performance seemed to reach a peak around a budget of 50 000 simulations.

Furthermore, the performance in pseudo-reality improved initially with an increased budget. Here, however, the performance leveled after the budget of 25 000 simulations was reached and did not improve any further. This could be an indicator that at that moment the design process reached the peak performance

that would still transferable to real robots. Further designs might improve the performance in simulation but the transferability will suffer in return.

Influence of the sample size

Figure 4.7 shows the results of the three different variants of the sample size over the three investigated budgets. For a budget of $25\,000$ simulations, all variants performed similar and no differences can be seen, both in simulation and pseudo-reality. In the case of a budget of $50\,000$, the variant with a sample size of 10 samples performed slightly better than the other two variants, in the mission FORAGING when assessed in simulation. In pseudo-reality, this difference however was not present anymore. It could therefore very well be that this was simply a statistical artifact of the stochastic design process. For $100\,000$ simulation runs, the three variants achieved a comparable performance again and only minor differences could be observed. All in all, the three different sample sizes that we compared showed no noticeable differences.

Influence of the restarting mechanism

Figure 4.8 shows the results of the different restarting mechanisms. The results for a budget of $25\,000$ simulation runs showed no difference between the four variants. In case of a budget of $50\,000$ simulation runs, all variants performed similarly in the mission AAC. In the mission FORAGING, the restarting mechanism that restarts every 5000 simulation runs performed worse than the other three variants. For a budget of $100\,000$ simulation runs, all four variants performed similarly again. In the mission FORAGING, however, the fixed length restarting mechanism (default) showed a larger variance than the other three variants.

In conclusion, the four different variants failed to produce noticeable differences in the performance of the generated control software.

Comparison with `Chocolate`

Figure 4.9 shows the comparison results of `IcePop` with `Chocolate` for budgets of $25\,000$, $50\,000$, and $100\,000$ simulations, respectively. Throughout all three budgets, it is apparent that `IcePop` performed better in simulation than `Chocolate` in both missions. In the mission AAC, the difference in performance is statistically significant.

Unfortunately the drop of performance when assessed in pseudo-reality was slightly larger for `IcePop` than for `Chocolate`. This could indicate that `IcePop` might be less transferable to real robots than `Chocolate`. Despite the larger

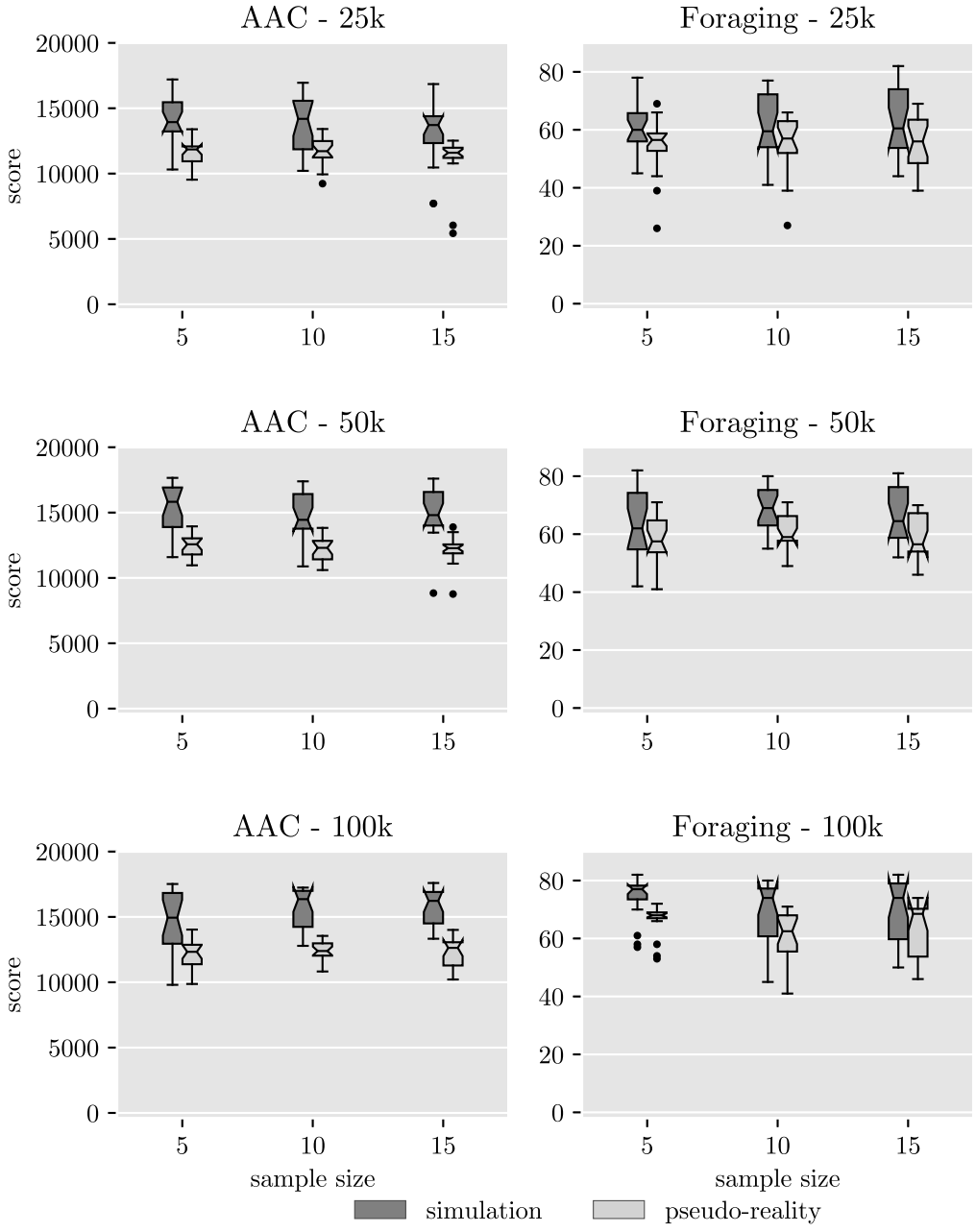


Figure adapted from Kuckling et al. (2020b).

Figure 4.7: Influence of the sample size on the performance of `IcePop`. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

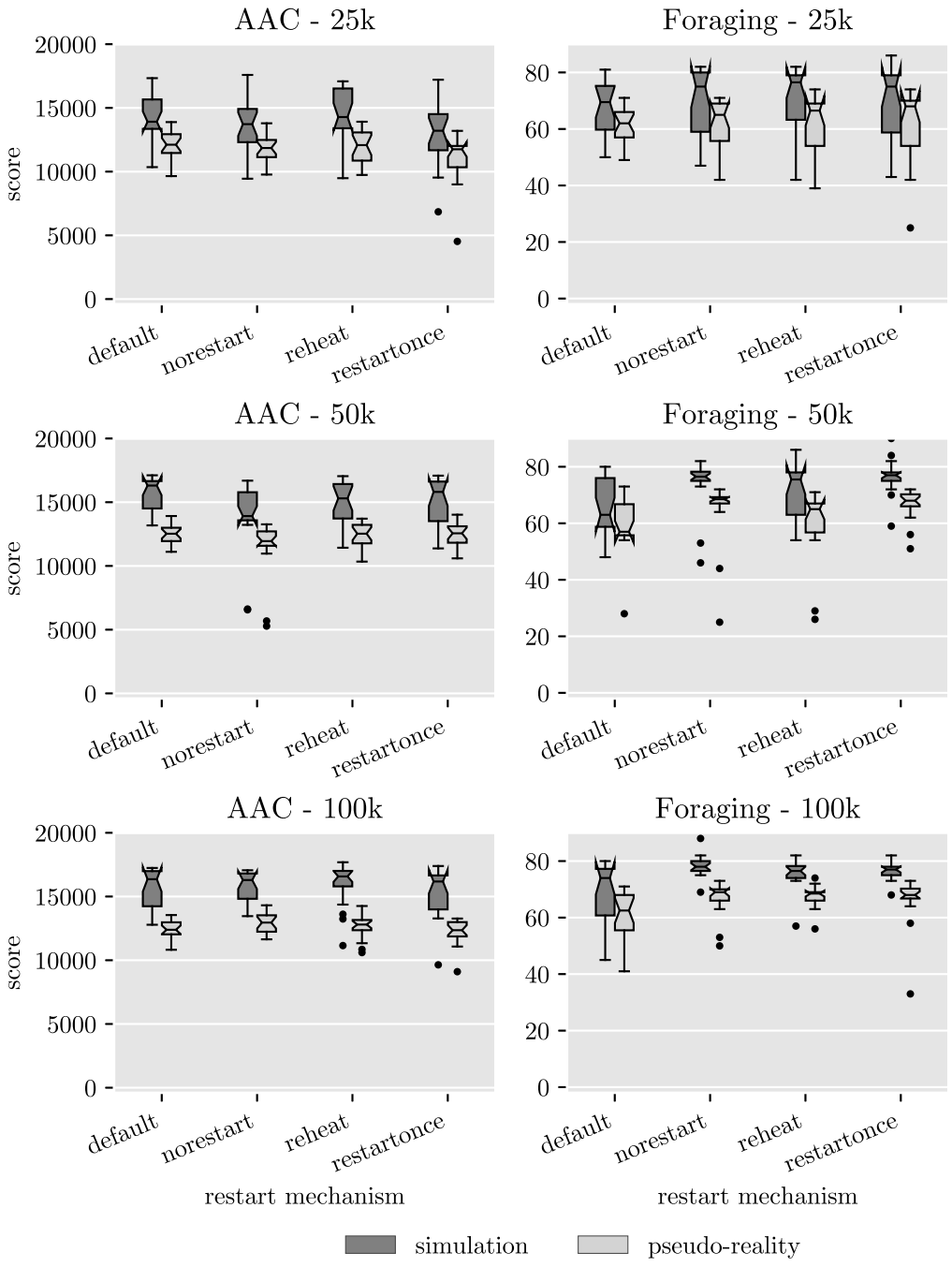


Figure adapted from Kuckling et al. (2020b).

Figure 4.8: Influence of the restart mechanism on the performance of `IcePop`. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

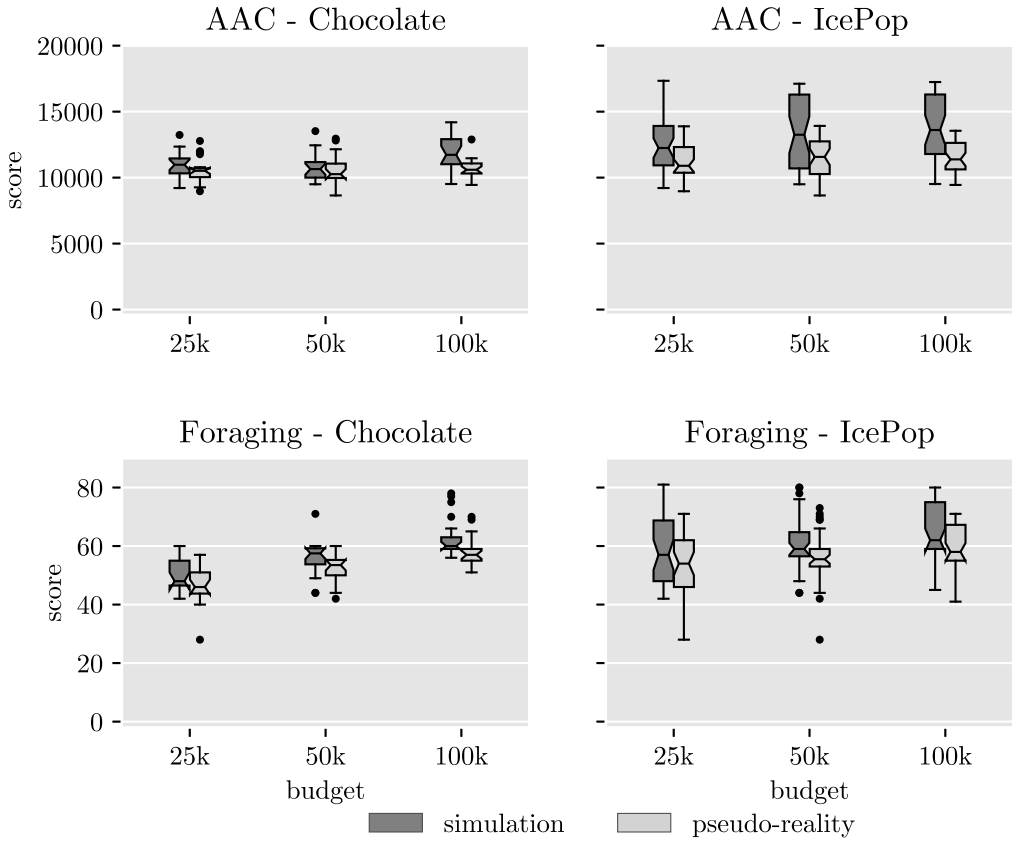


Figure adapted from Kuckling et al. (2020b).

Figure 4.9: Comparison between `Chocolate` and `IcePop`. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

performance drop, IcePop still performed better in pseudo-reality, and in AAC this difference in performance was also statistically significant.

Additionally, we have taken the best performing instance of control software of IcePop and Chocolate (with a design budget of 100k simulations) for each mission and directly applied it to a swarm of twenty real e-pucks. We did not collect statistical data about the performance, but visual inspection showed that IcePop transferred similarly well into reality as Chocolate. Videos of the performance of the control software on real robots can be found online (Kuckling, 2023).

4.4.3 Discussion

The results indicate that simulated annealing can be a viable candidate for the automatic modular design of robot swarms. Additionally, I have investigated the influence of some obvious variations to the simulated annealing on the performance of the automatic modular design. Neither variation provided a consistent advantage over the chosen defaults. However, the component-based simulated annealing approach allowed to easily implement these variants.

4.5 Limitations and possible improvements

The work presented in this chapter has shown that local search algorithms are viable candidates as optimization algorithms in the automatic modular design of control software. Neither Cherry nor IcePop performed worse than Chocolate, and even outperformed it for some combinations of mission and budget. However, a few areas of possible improvement remain:

The search space is not well understood. The control software generated by Cherry and IcePop performed no worse than the one generated by Chocolate, which is a design method known to avoid local optima. Therefore, it seems appropriate to conclude that the search space of the considered missions allowed the local search-based design methods to converge towards either the global optimum or at least towards local optima that were not significantly less successful than the global one. It is, however, unclear if this was only the case because I considered relatively simple missions, or if the search space of swarm robotics missions is well-behaved in general. Further research on the analysis of search spaces is therefore necessary.

The definition of the neighborhood can impact the performance of the local search. Another confounding factor is the definition of the neighborhood. Lo-

cal search algorithms require the definition of a neighborhood function, which describes the way in which they traverse the search space. A poorly defined neighborhood might create several local optima from which escape is impossible. The neighborhood that I defined in this work was conceived arbitrarily, based on my intuitions and without prior evidence of its suitability. In order to port the design methods to other control architectures, such as behavior trees, a new neighborhood function must be defined. Further research would therefore be necessary to better understand how neighborhoods can be properly defined. At the moment, this limits the general applicability of local search algorithms to the automatic modular design of control software for robot swarms.

Application of other algorithms and automatic algorithm design. Lastly, this work has focused on two relatively simple local search algorithms. However, several other local search algorithms have been proposed in the literature. Choosing the correct optimization algorithm might alleviate the previously mentioned limitations, should they arise. Furthermore, following the component-based approach of Franzin and Stützle (2019), it might be possible to perform automatic algorithm design to select the most appropriate algorithm and its components for the automatic modular design of control software for robot swarms.

Chapter 5

Model-based optimization algorithms

To the best of my knowledge, all optimization algorithms studied in the automatic design of control software for robot swarms are model-free. That is, the algorithms have no model of the search space and can only estimate the quality of a candidate solution by performing an evaluation. Model-based optimization algorithms, contrarily, maintain a model of the search space. With this model they can estimate the quality of a candidate solution but evaluations remain necessary to validate the estimation and to possibly update the model. Model-based optimization algorithms therefore promise to make better use of the limited number of evaluations available to them. I developed `Schwarzwälder2` and `Schwarzwälder3`, two automatic modular design methods that are based on two implementations of SMAC, a model-based optimization algorithm.

5.1 SMAC

SMAC (Hutter et al., 2011; Lindauer et al., 2022) (sequential model-based algorithm configuration) is a model-based metaheuristic optimization algorithm that has been successfully applied to configure several algorithms (Kerschke et al., 2019). It combines an aggressive racing scheme with a surrogate model for generating new instances of control software. In the aggressive racing scheme, SMAC maintains an incumbent candidate solution and compares it against a challenger candidate solution. Inferior challengers are often discarded after a single evaluation and more evaluations are only performed when the challenger could replace the incumbent. The surrogate model is a random forest of regression trees and is built from all previous evaluations. New challengers are sampled using the random forest model. SMAC2 is an implementation of the original SMAC algorithm in Java (Hutter

et al., 2011). SMAC3 is a reimplementation of the SMAC algorithm in Python with slightly different default settings (Lindauer et al., 2022).

5.2 Design methods

I developed two variants of `Chocolate`—AutoMoDe-`Schwarzwälder2` and AutoMoDe-`Schwarzwälder3`—that are based on SMAC2 and SMAC3, respectively.[1] These variants differ only in the choice of the optimization algorithm, namely SMAC2 and SMAC3. All other elements of the design methods are equal. That is, `Schwarzwälder2` and `Schwarzwälder3` generate control software for the e-puck robot, as defined by RM1.1. The control software is assembled into finite-state machines with up to four states and up to four outgoing transitions per state. The modules available to `Schwarzwälder2` and `Schwarzwälder3` are the modules used in `Chocolate`. The optimization algorithm is free to choose the number of states and transitions within the constraints, the modules to associate with the states and transitions, and the value of the parameters of chosen modules, where applicable.

Unlike Iterated F-race, which samples an initial population randomly, SMAC2 and SMAC3 require to specify the initial incumbent. In this work, I chose the parameters of the initial incumbent to be the lower bound of all allowed intervals. Categorical parameters were mapped to integer values and the lowest integer value was chosen as the value for the initial incumbent. The such defined incumbent encoded a finite-state machine that is composed of one state that executes the exploration behavior.

5.3 Experiments

I validated `Schwarzwälder2` and `Schwarzwälder3` by performing experiments on thirty randomly generated missions. The generated control software was evaluated both in simulation and on real robots.

5.3.1 Missions

I randomly generated thirty missions, using the mission generator MG1 proposed by Ligot et al. (2022). Ten missions had the objective of HOMING, ten the one of

[1] All flavors of AutoMoDe have names related to food items. In that tradition, the names of the methods presented here are chosen in reminiscence of the Schwarzwälder Kirschtorte (black forest cake). It has been chosen to highlight that the optimization algorithms for these design methods are based on random forests.

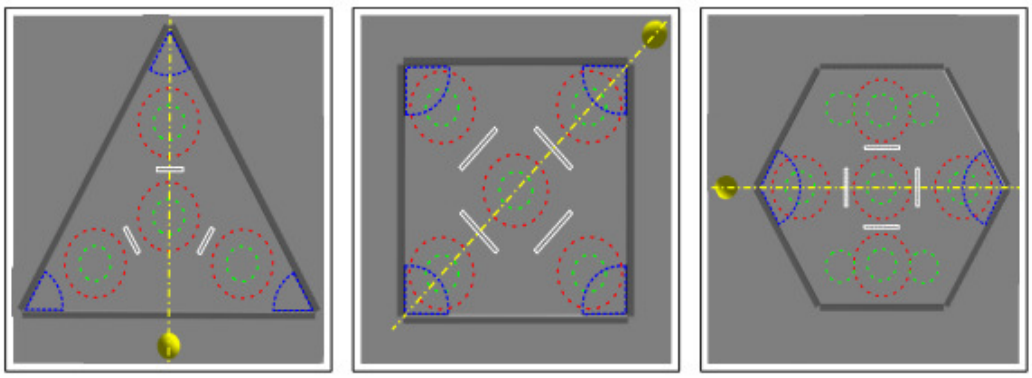


Figure originally published by Ligot et al. (2022).

Figure 5.1: Possible arena layouts that the mission generator MG1 can generate. The mission generator was allowed to choose the arena shape and, depending on the shape, it could place obstacles or floor patches in the arena. Red, blue, and green circles show possible locations for circular floor patches. White rectangles indicate the possible locations for the obstacles. The yellow line denotes the light axis used to place specific patches in the missions of type FORAGING.

AGGREGATION and ten the one of FORAGING. In HOMING, the swarm was tasked to aggregate on a specific area indicated by either white or black floor (chosen by the mission generator). Areas of different color might serve as a distraction or a cue to the swarm. In AGGREGATION, the swarm was tasked with selecting and aggregating on a single area out of multiple available candidates. Areas of different color might serve as a distraction or a cue to the swarm. In FORAGING, the swarm was tasked with an abstracted foraging task in which the robots must transport items from food sources to the nest area. When moving over a food source, the robot automatically collected a virtual food item and, when moving over the nest area, the virtual food item was automatically deposited in the nest.

Each type of mission took place in a randomly generated arena (see Figure 5.1 for potential layouts of the arena). For each mission, the swarm size was randomly chosen to be either 15 or 20 robots. Similarly, the experiment duration of each mission was randomly chosen as either 60 s, 120 s, or 180 s.

5.3.2 Protocol

I compared the original version of `Chocolate` (Francesca et al., 2015) with `Schwarzwälder2` and `Schwarzwälder3`. For reference, I also included in the experiments `EvoStick` (Francesca et al., 2014), a yardstick implementation of neuro-evolutionary swarm robotics (see Section 3.5).

I ran each design method on each mission with five different design budgets: 5000, $10\,000$, $20\,000$, $50\,000$, and $100\,000$ simulation runs. Simulations were performed using the ARGoS3 simulator (Pinciroli et al., 2012) with a realistic noise setting (see Section 3.6). For each combination of design method, mission, and budget, I ran a single design. I ran each optimization algorithm with the default values of their implementation. I collected the generated instances of control software and assessed them once in simulation. Additionally, I assessed the performance of all instances of control software that were generated for a budget of $100\,000$ simulation runs on the real robots.

Because the resulting scores span different orders of magnitudes (10^{-1} to 10^{5}) depending on the choice of mission and objective function, I present the results after normalization. The normalized score $s'_{i,j}$ for design method j in mission i was computed as $s'_{i,j} = (s_{i,j} - min_i)/max_i$, where $s_{i,j}$ was the actual score obtained and max_i and min_i were the maximum and minimum scores obtained in reality for mission i. Thus, all scores in reality were scaled in the interval $[0, 1]$, while scores in simulation might be higher. This normalization procedure was previously adopted by Hasselmann et al. (2021, 2023).

I present the results in the form of notched boxplots and Friedman plots (see Section 3.7).

5.4 Results

Figure 5.2 shows the development of the performance in simulation, for all design methods and budgets under consideration. For `Chocolate` and `Schwarzwälder2`, the quality of the solutions generated improved with increasing budgets, but seemed to reach a plateau with a design budget of $50\,000$ simulation runs, with no clear improvement for a budget of $100\,000$ simulation runs. For `Schwarzwälder3` and `EvoStick`, the quality of the solutions generated improved with increasing budgets without reaching a plateau.

Figure 5.3 shows the performance in simulation and reality for a budget of $100\,000$ simulation runs. When comparing the normalized scores achieved by the design methods per budget, all four methods appear to have achieved similar performance in simulation. Concerning the performance in reality, `EvoStick` suffered most strongly from the reality gap. The three modular design methods appeared to have been less affected by the reality gap.

Figure 5.4 shows Friedman plots of the results obtained in simulation and in reality by all design methods for a budget of $100\,000$ simulation runs. In simulation, `Schwarzwälder3` and `EvoStick` outperformed `Chocolate` and `Schwarzwälder2`. When assessed in reality, `EvoStick` was outperformed by all other design methods. This rank inversion has been often observed when

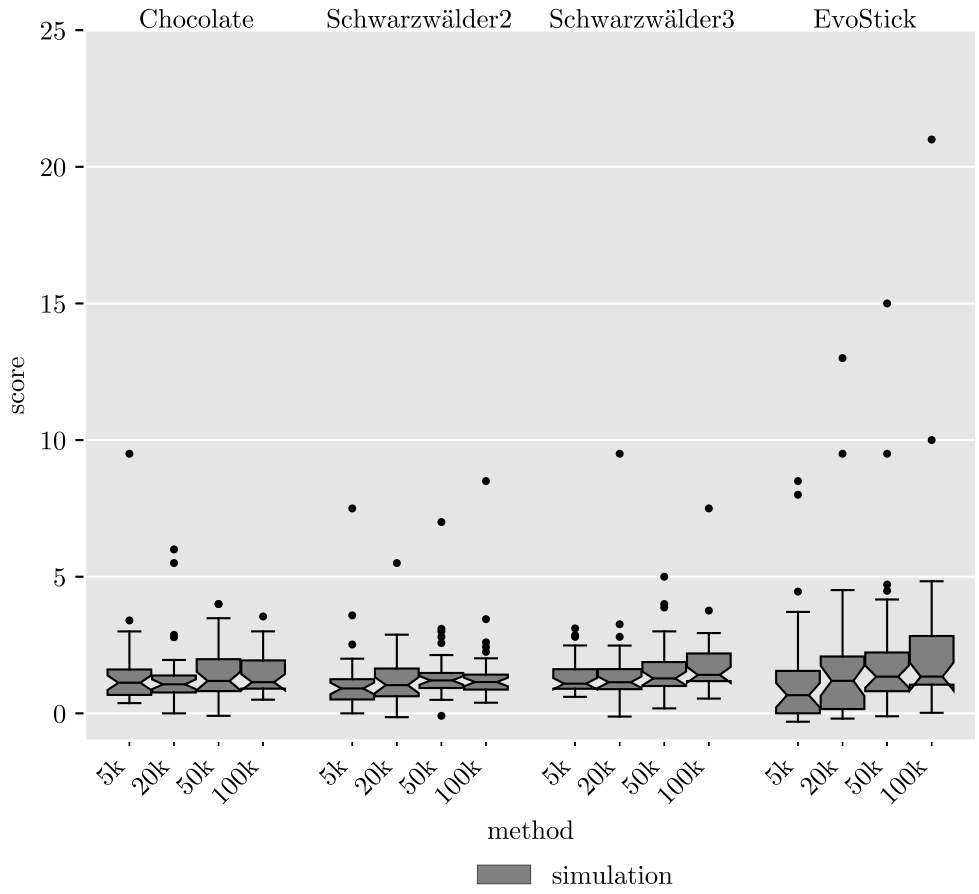


Figure adapted from Kuckling et al. (2023).

Figure 5.2: Boxplots of normalized performance of all considered design methods for all considered budgets. All results were obtained in simulation and normalized according to the experimental protocol.

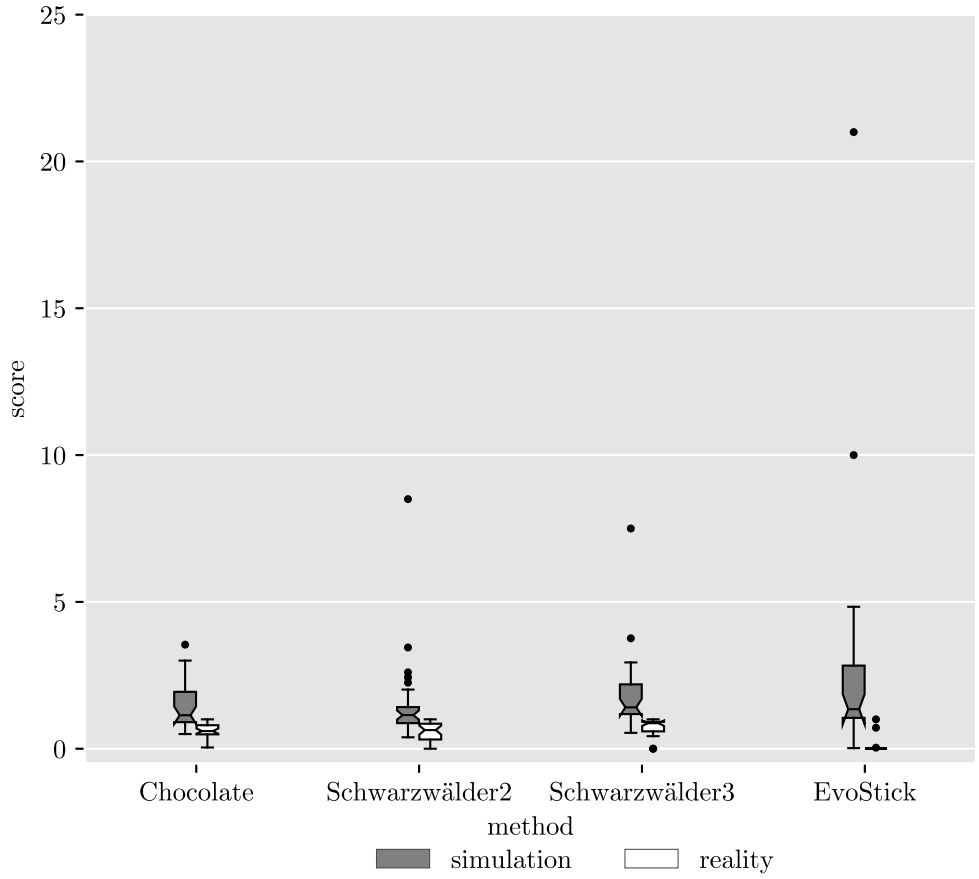


Figure adapted from Kuckling et al. (2023).

Figure 5.3: Boxplots of normalized performance of all considered design methods with a budget of $100\,000$ simulation runs. Results were obtained in simulation and in reality and normalized according to the experimental protocol. The dark grey boxes represent performance obtained in the design context, white boxes represent performance obtained in reality.

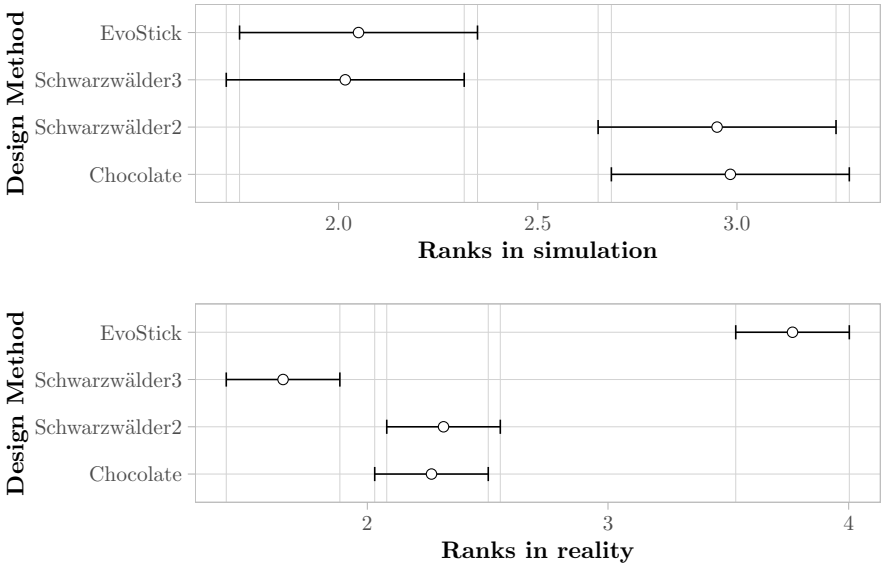


Figure originally published in Kuckling et al. (2023).

Figure 5.4: Friedman plots of average ranks of all considered design methods for a design budget of 100 000 simulation runs. Ranks are shown for simulation (top) and reality (bottom).

Table 5.1: Average solution quality of all design methods. All scores were obtained in reality and normalized according to the experimental protocol.

	mean	95% conf. int.	std. dev.
`Schwarzwälder3`	0.75	[0.65, 0.85]	0.27
`Chocolate`	0.62	[0.54, 0.71]	0.23
`EvoStick`	0.06	[-0.02, 0.14]	0.22

neuro-evolutionary design methods were compared with modular design ones. Here, it is worth noticing that the three modular methods maintained their relative ranking when tested on the robots. This is an indicator that, while the optimization algorithm might have an impact on the performance of automatic modular design methods, it does not appear to significantly affect their ability to cross the reality gap.

To further study the quantitative differences between the design methods, I investigated the average performances of `Chocolate`, `Schwarzwälder3` and `EvoStick`. Table 5.1 reports relevant statistics of the distribution of the normalized scores obtained on all the missions. Based on the mean performance, `Schwarzwälder3` offered a relatively small improvement over `Chocolate`:

Table 5.2: Differences in solution quality between pairs of design methods. All scores were obtained in reality and normalized according to the experimental protocol.

	mean	95% conf. int.	std. dev.
Schwarzwälder3 - Chocolate	0.12	[-0.02, 0.26]	0.37
Schwarzwälder3 - EvoStick	0.69	[0.55, 0.82]	0.36
Chocolate - EvoStick	0.57	[0.43, 0.70]	0.36

both Schwarzwälder3 and Chocolate provided a much larger improvement over EvoStick. Table 5.2 reports statistics on the differences of the scores obtained by Schwarzwälder3, Chocolate, and EvoStick. Indeed, the mean difference between Schwarzwälder3 and Chocolate was relatively small when compared to the one between Chocolate and EvoStick, which was about five times larger than the former.

The results indicated that Schwarzwälder3 was the best performing design method. When assessed in reality, it improved over Chocolate and outperformed EvoStick. In order to better understand the nature of the differences in performance, I compared the scores obtained in simulation by Chocolate, Schwarzwälder3 and EvoStick on a per mission basis. By studying the performance in simulation, I could investigate the designed behaviors in their original context, regardless of the effects of the reality gap on the actual behavior. Figure 5.5 shows the performance in simulation of the three selected design methods for all thirty missions. Chocolate and Schwarzwälder3 showed similar performance on most missions. Overall, Schwarzwälder3 outranked Chocolate in twenty four of the thirty missions, yet the difference in performance remained relatively small, except for a few missions such as mission 8 and 30. Schwarzwälder3 outranked EvoStick in eighteen of the thirty missions, whereas EvoStick outranked Schwarzwälder3 in the other missions. Yet, the difference in performance was larger than between Chocolate and Schwarzwälder3.

To quantify the extent to which the performance of two design methods differs, I considered the absolute percentage deviation given by $\frac{|x-y|}{min(x,y)} \cdot 100\%$. This measure does not consider which design method performed better, but describes the magnitude of the differences observed. The mean absolute percentage deviation between the performance of Chocolate and Schwarzwälder3 was 37%. When comparing Schwarzwälder3 and EvoStick, contrarily, the difference in the observed performance was relatively large—the mean absolute percentage deviation was 131%. However, this deviation was caused mainly by one outlier in mission 24, where EvoStick obtained a score twenty times higher than

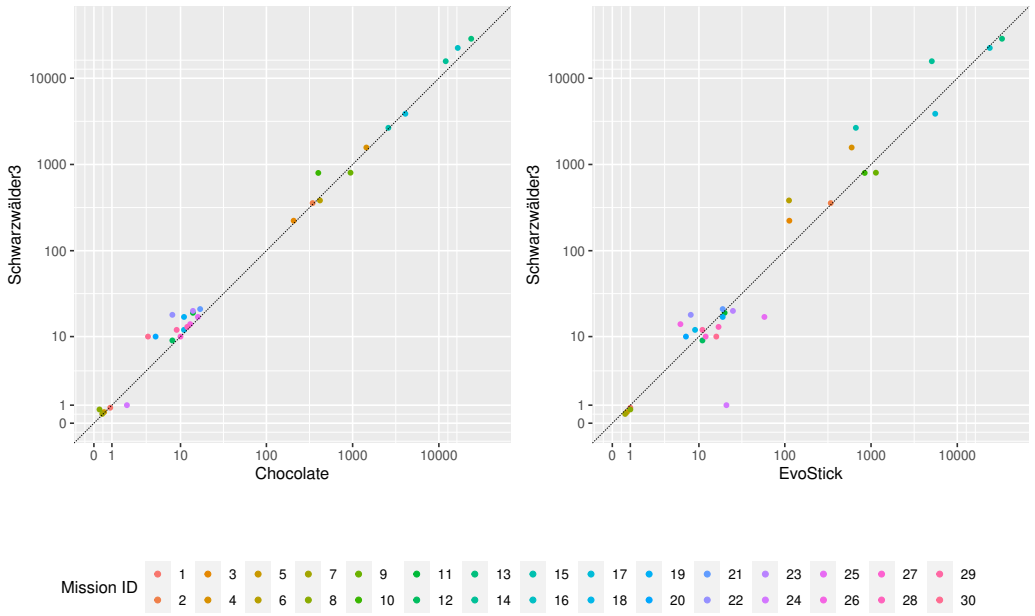


Figure originally published in Kuckling et al. (2023).

Figure 5.5: Scatterplots showing performance in simulation for a design budget of 100 000 simulation runs. Left: `Chocolate` and `Schwarzwälder3`, right: `Schwarzwälder3` and `EvoStick`. Scores are plotted on a logarithmic scale.

`Schwarzwälder3`. Ignoring the outlier, the mean absolute percentage deviation was 69%.

In conclusion, `Schwarzwälder3` outranked `Chocolate` more often than it outranked `EvoStick`, yet the mean absolute percentage deviation between `Schwarzwälder3` and `EvoStick` was larger than between `Schwarzwälder3` and `Chocolate`. I hypothesized that this might be explained by the fact that `Chocolate` and `Schwarzwälder3` designed finite-state machines from the same set of modules, whereas `EvoStick` designed artificial neural networks. `Chocolate` and `Schwarzwälder3` found therefore similar behaviors, but in general `Schwarzwälder3` optimized these behaviors better than `Chocolate` did. Conversely, `EvoStick` produced control software in the form of a different architecture and therefore generated behaviors that are considerably (and structurally) different from those produced by `Schwarzwälder3`. As a result, a good behavior found by `EvoStick` might have clearly outperformed the one found by `Schwarzwälder3`, and vice versa.

To further support this hypothesis, I investigated the choice of modules by the

Figure originally published in Kuckling et al. (2023).

Figure 5.6: Plots of time spent executing each module per mission for the instances of control software designed for a budget of 100 000 simulation runs.

modular design methods. Interestingly, `Schwarzwälder2` and `Schwarzwälder3` generated instances of control software that used more states and transitions than `Chocolate`. More important than the selection was the use of the selected modules—e.g., a selected module might never be invoked. Figure 5.6 shows the time spent in each behavioral module during the execution of the evaluation in simulation. For most missions, all modular design methods appeared to have relied on similar behaviors, resulting in a similar amount of time spent per module. Only in a few missions, the modular design methods appeared to have generated different behaviors. These missions with different behaviors also seemed to coincide with those that showed large differences in the performance between `Chocolate` and `Schwarzwälder3`. Per the experimental protocol, I only ran a single design per mission. Therefore, this experimental protocol is not suited to answer to what extent these cases of differing behaviors are an indicator of the ability or inability of `Chocolate` and `Schwarzwälder3` to reliably generate these differing behaviors for a given mission. However, the experimental protocol did allow to conclude that `Schwarzwälder3` was more likely than `Chocolate` to generate them. The

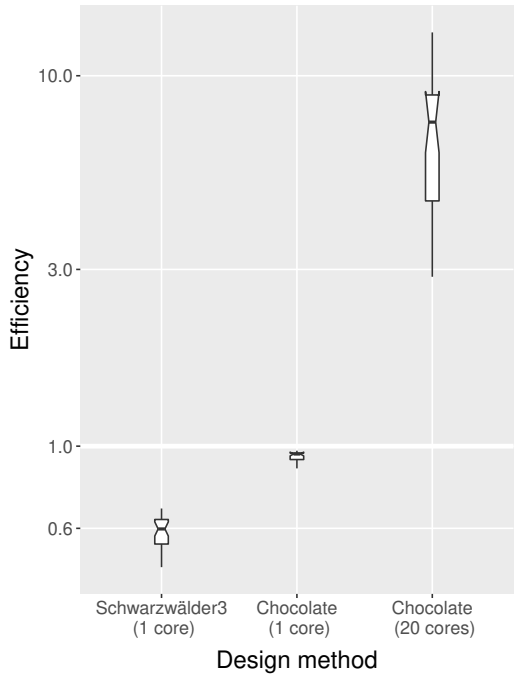


Figure originally published in Kuckling et al. (2023).

Figure 5.7: Run time efficiency of `Chocolate` and `Schwarzwälder3`, computed as the ratio of time spent running only the simulations (CPU time) with respect to total run time (wall time). All experiments were performed on a high-performance computing cluster equipped with Epyc7452 CPUs.

fact that most missions used the same set of modules further supported my claim that the differences between `Chocolate` and `Schwarzwälder3` are due to a better tuning of the control software rather than different strategies employed. One reason could be that SMAC is a model-based optimization algorithm, as opposed to Iterated F-race. This could allow `Schwarzwälder3` to sample better challengers from the learned model.

5.5 Runtime analysis

It is widely understood that the performance of a design method is a function of the resources available. Any comparison between design methods, to be fair and meaningful, must be done by putting the methods under analysis on the same foot for what concerns the resources made available to them. Imposing a design budget, to cap the available resources, is a common approach in automatic design of control software for robot swarms, although sometimes it is done indirectly through the

limitation of the number of generations or epochs—e.g., see Trianni (2008). As part of the experimental protocol, I followed this convention and defined a budget of simulations that should not be exceeded by the design process.

Studying the run time of the design methods under analysis was not one of my original concerns and, therefore, the experimental protocol I defined and adopted did not provide for gathering data that allowed addressing this issue. Yet, I performed some further experiments by running `Chocolate` and `Schwarzwälder3` on a high-performance computing cluster, equipped with Epyc7452 CPUs,[2] with a design budget of $100\,000$ simulation runs. In the original designs, `Chocolate` was parallelized using 20 computational cores. However, our chosen implementation of SMAC3 does not provide the ability to parallelize the design process across multiple cores[3]. Therefore, I executed `Schwarzwälder3` on a single computational core. I wished to compare `Chocolate` and `Schwarzwälder3` under similar setups to investigate whether the difference was just because of the parallelization. Therefore, I also measured the run time of `Chocolate` when only one core was allocated.

For `Schwarzwälder3`, the mean time to complete one design was 50 hours and 33 minutes. Conversely, for `Chocolate`, executed on a single core, the mean time to complete one design was 26 hours and 18 minutes. It is noteworthy that `Chocolate`, running on a single core, finishes its designs in about half the time needed by `Schwarzwälder3`. To further illustrate this discrepancy, I compared the computational overhead of the design methods, that is, the time that a design method spends outside of performing simulations. It is to note that a rigorous analysis of the run times was beyond the scope of the work presented in this dissertation. I simply provide some first insight on the run time, as I observed it by following the experimental protocols adopted in this work. Figure 5.7 reports the run time efficiency of `Schwarzwälder3` and `Chocolate`. I computed the run time efficiency as $t_{total}/t_{simulation}$, where t_{total} was the total wallclock time that a design process needed to finish and $t_{simulation}$ was the total CPU time spent executing simulations. In the case of a design run on a single core, the efficiency therefore described the proportion of time spent performing simulations. When executed on a single core, `Chocolate` had a mean efficiency of 0.94, that is, on average, the whole design process spent 94% of the time performing simulations. `Schwarzwälder3`, contrarily, had a mean efficiency of 0.59, meaning that it only spent 59% of the time performing simulations. This is because SMAC3 is a model-based optimization algorithm, whereas Iterated F-race is not. Therefore, SMAC3 needed to maintain the model built from the previously evaluated instances

[2] `https://www.amd.com/en/products/cpu/amd-epyc-7452`

[3] However, alternative protocols parallelizing the design process by running independent partial designs are in principle possible.

of control software and their performance. With an increasing number of previous instances of control software, the complexity of the model increased and it required a non-trivial amount of time to update the model. Iterated F-race, conversely, required very little time outside of simulations. Yet, the model appeared to have allowed `Schwarzwälder3` to find better fine-tuned control software than `Chocolate` could produce.

5.6 Discussion

I tested the four design methods considered on thirty automatically generated missions. When assessed in simulation, `EvoStick` and `Schwarzwälder3` outperformed `Chocolate` and `Schwarzwälder2`. When assessed in reality, I observed a rank inversion between `EvoStick` and the modular design methods. Results showed, however, that all modular design methods maintained their relative ranks when assessed in reality. This indicated that the considered modular design methods are inherently robust to the reality gap and that the robustness does not seem to be affected by the optimization algorithm.

In an attempt to explain the performance difference between `Schwarzwälder3` and `Chocolate`, I investigated the generated behaviors. All modular design methods generated instances of control software that produced similar strategies, Therefore, I attributed the difference in performance to the quality of the tuning of the parameters, rather than behavioral differences.

The observations made about the run time raise some concerns about the limitations of the employed experimental protocol. In a simplified manner, an automatic design method can be described as the repetition of two steps: generating new instances of control software and evaluating them. The computational effort of an automatic design method can then be seen as the sum of the computational efforts for each step. Prior to our experiments, most automatic design methods were structurally similar due to the fact that the computational effort required for evaluations strongly dominated the computational effort required for generating new instances of control software. As a result, the standard experimental protocol in swarm robotics is built on the unspoken assumption that the computational effort of generating instances of control software can be neglected. Consequently, the standard experimental protocol ensures that design methods are compared with similar computational resources by imposing a limit on the amount of available evaluations. `Schwarzwälder3` spent a significant amount of computational effort to maintain its model to generate new instances of control software from it, thus violating the assumption underlying the standard experimental protocol.

In order to develop a new experimental protocol that takes computational effort for generation and evaluations into account, it is necessary to find answers

to two important questions: *How can we quantify the computational efforts expended for each part? And, how can we ensure that different implementations and experimental setups remain comparable with each other?*

How can we quantify the computational efforts expended? Computational effort is commonly quantified related to the expenditure of time. While this could be a direct measurement of the time (as often done in optimization research), it could also be an abstract quantity representing a unit of expended time (for example, the number of operations, as underlying the computational complexity classes). The naive way to quantify expended time would be to measure the time between the beginning and the end of the design process (*wall time*). The wall time is easy to measure and often corresponds to real-world constraints. However, it does not accurately reflect the amount of parallel resources spent and might be impacted if the design process is interrupted or suspended by other processes running on the same machine. An alternative is *CPU time*, which measures the total time spent across all CPU cores.

Commonly, also an abstract representation of time is used, for example, the *number of operations*. The abstract units of time are usually chosen in such a way that they represent an overwhelming majority of the computational effort. In the case of swarm robotics, a simulation was usually chosen as the tracked operation. With a negligible amount of computational effort to generate new instances of control software, one could express the total used computational resources as the number of allocated simulations. For measuring the computational effort of generating new control software in an abstract way, one would need to find a unit of quantity, similar to "one simulation". However, this unit would need to correspond to an arbitrary but fixed amount of computational effort, which might not be possible as often the computational effort of an operation on a surrogate model is not constant but varies in the size of the model.

Another quantity that could be measured is *memory*, representing the size of the model (if any) used to generate new instances of control software. Similarly to time, memory can either be measured directly or in abstract quantities. Maintaining a large model would then equate to a large expenditure of computational resources, whereas small or no models expend less computational resources. However, memory is often not a limiting factor in the automatic design of control software for robot swarms.

Another element of computational effort can be the amount of *allocated hardware*, such as CPUs or GPUs. Again, a naive approach could be to measure the allocation, but this might overlook cases in which more hardware is allocated than can be efficiently used by the automatic design method. It would therefore be necessary to find a more complex quantification that correctly accounts for

allocation and use of different computational hardware.

How can we ensure that different implementations and experimental setups remain comparable with each other? Regardless of which quantity is used to measure the computational effort, it is necessary to define the computational effort in such a way that experimental setups remain comparable. A measurement of the computational effort should at the very least make different runs of the same design method and under the same protocol comparable to each other. This means that the measurement should either be defined to be hardware-independent or account for the choice of computational hardware. For example, if the computational effort is naively measured as wall time, the computational effort depends on the chosen computational hardware. A design method run on older hardware would be able to perform less computations in the same wall time as the same design method run on newer hardware. Consequently, this would hinder the reproducibility of experiments from the literature, as each experiment needs to be run on the exact same hardware it was originally conceived for.

Another major challenge is the comparison across different design methods. In this case, it is possible that not only the chosen computational hardware but also the efficiency of the implementation impacts the measurement of expended computational effort. This holds especially in a research setting, in which it is common that the developers of the implementation rarely spent the time to optimize it. Thus, it would become impossible to compare a prototyped and poorly optimized implementation of a design method with an established and well-optimized one.

5.7 Limitations and possible improvements

The work presented in this chapter has shown that model-based optimization algorithms are another viable candidate as optimization algorithm in the automatic modular design of control software for robot swarms. Furthermore, all modular design methods generated instances of control software that produced control software that crossed the reality gap satisfactorily. However, I also observed that the quality of the tuning of model-based optimization algorithms came at the expense of the run time of the design process. Moreover, a few areas of possible improvement remain:

Choice of incumbent: Unlike Iterated F-race, SMAC requires the definition of an initial incumbent candidate solution. In `Schwarzwälder2` and `Schwarzwälder3`, I have selected the incumbent arbitrarily through the lowest possible value of the representation of all parameters. I did not study the effect that this choice had on the performance of `Schwarzwälder2` and `Schwarzwälder3`.

From my results obtained with `Cherry`, it appears that the initial incumbent candidate solution did not have a strong effect on the performance of the design method but I believe that further research is required. Possibly, this could lead to a hybridization of Iterated F-race and SMAC. Iterated F-race has shown to produce viable instances even at low design budgets, albeit they perform worse than control software generated with higher design budgets. Therefore, a small fraction of the design budget might be used to generate such an instance of control software with Iterated F-race and use it as the initial incumbent in SMAC to design the final instance of control software.

Automatic algorithm configuration: SMAC, just like Iterated F-race, provides many parameters that influence its performance. While the default values of both algorithms have shown to produce promising results, it is not clear if this was the best possible performance for either algorithm. As manual algorithm tuning is infeasible due to the size of the search space, it could be interesting to perform an automatic configuration of the parameters of either optimization algorithm.

Improved experimental protocol: The results of this work indicated the need to refine the experimental protocol for the automatic design of control software for robot swarms. The standard experimental protocol defined the stopping criterion of the design process by specifying a design budget. I observed that the model maintained by SMAC3 improved the performance of generated control software at the expense of increased run time. A new stopping criterion should therefore not only consider the number of simulations, but also the overall computational effort of the design process, in order to allow a comparison on equal grounds. However, defining a way to measure the run time is not straightforward (see Section 5.6).

Furthermore, the question of parallelism poses another challenge. Some design methods, like `Chocolate` can easily be highly parallelized. Other design methods, like `Schwarzwälder3`, cannot be parallelized out-of-the-box. However, it is an open question if the allocation of the whole budget to a single run of the optimization algorithm is optimal. Instead, it could be possible to allocate only a fraction of the design budget to a number of independent runs of the optimization algorithm. The final instance of control software is then the best performing instance out of the returned set. Following this protocol, it would be possible to also parallelize sequential optimization algorithms, by parallelizing the independent runs.

Chapter 6

Perspective on optimization-based design

As discussed in Section 2.4.3, optimization-based design can be classified according to several criteria. Due to the limitations outlined in Section 2.5, online design is not yet a viable design paradigm in the optimization-based design of control software for robot swarms. Conversely, offline design methods have shown to generate well-performing instances of control software in many settings (see Section 2.6). Despite the many differences, several similarities remain among optimization-based offline design methods.

In this chapter, I present my perspective on optimization-based offline design. In particular, I argue that optimization-based offline design might be conceived at three different levels (see Figure 6.1). Each level encompasses several classes of design methods that share the same conceptual approach to optimization. The three levels are: *Tuning*, *Assembling*, and *Shaping*.

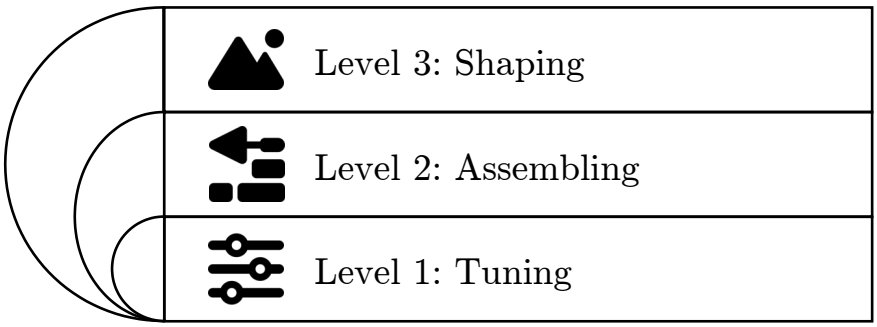


Figure 6.1: The three levels of optimization-based design of control software for robot swarms. Each higher level encompasses all underlying ones.

6.1 Level 1 - Tuning

At this first level, optimization-based offline design is concerned with optimizing the free parameters of a partially defined instance of control software. Typically, a design method on this level might start from a manually designed baseline instance of control software and then automatically tunes its free parameters. For example, Ijspeert et al. (2001) studied a cooperative stick-pulling experiment. They designed a finite-state machine that controlled the robots and systematically investigated the influence of different parameters on the performance. Hecker et al. (2012) designed a finite-state machine from the observation of the foraging behavior of ants. Using a genetic algorithm, they optimized several parameters of the behaviors. Ligot et al. (2020b) created control software in the form of finite-state machines by manually defining a simple solution and tuning its parameters using Iterated F-race. These finite-state machines were used as baselines to compare against more complex automatic modular design methods. However, the partial instance of control software does not necessarily need to be a finite-state machine. Another typical design pattern is physicomimetics (Spears et al., 2005; Spears and Spears, 2012), a pattern to design physics-inspired control software for robot swarms. For example, Dimidov et al. (2016) exhaustively searched the parameter space of Lévy walks and correlated random walks to find optimal parameter configuration in a search mission for bounded and unbounded environments. Engebråten et al. (2020) created a behavioral repertoire using MAP-elites to find optimal parameter combinations of an artificial potential field. Design methods that start from a human-defined instance of control software are *semi-automatic* design methods, as they rely on the contribution of a human during the design process. Furthermore, they can only create control software for a single mission, as different missions would require different instances of control software as starting points.

Alternatively, one might specify a partial instance of control software that can be fine-tuned to address several missions. For example, this can be done by using neuro-evolutionary swarm robotics. In neuro-evolutionary swarm robotics (Trianni, 2008; Nolfi, 2021), the topology of an artificial neural network is defined. The design process than optimizes the weights of the neural network through the use of an evolutionary algorithm. Early works on evolutionary swarm robotics, such as the one of Trianni et al. (2003) and Dorigo et al. (2003), used feed-forward single-layer perceptron networks and optimized the weights of the connections. Similar approaches have been used in several works in the literature. For example, Gauci et al. (2014a) fixed the topology of a fully connected recurrent network and optimized the weights of the connections. Hasselmann et al. (2021) systematically investigated the design of single- and multi-layer perceptrons. In another research line, Gauci et al. studied the emergence of collective behaviors for robots with minimal capabilities. In their study, the robots only had a single line-of-sight sensor

and could set their velocity based on the discrete readings of this sensor. The authors used CMA-ES (Hansen and Ostermeier, 2001) to optimize the mappings of the sensor to velocities in missions such as clustering (Gauci et al., 2014b), shepherding (Özdemir et al., 2017; Dosieah et al., 2022), decision making (Özdemir et al., 2018), and coverage (Özdemir et al., 2019). Neuro-evolutionary design methods can be either *automatic* or *semi-automatic*, depending on whether a human designer remains active during the design process. However, as neuro-evolutionary design methods usually map all available sensors to all available actuators, the same fixed topology can be applied, in principle, to design control software for all missions of the class of all possible missions that is implicitly defined by the robotic platform.

6.2 Level 2 - Assembling

In the next higher level of our proposed hierarchy, design methods not only tune the free parameters of a partial instance of control software, but rather assemble the instance of control software from smaller, pre-defined elements—henceforth called modules. The modules are assembled into a higher-level structure, called the architecture. Common architectures include behavior trees and finite-state machines. However, is is usually the case that the modules have free parameters themselves. This allows to tune the modules to better interact with the rest of the assembled instance of control software. Therefore, Level 2 design also encompasses the concepts behind Level 1 design.

Several works in swarm robotics can be classified as Level 2 optimization-based design. For example, in GESwarm, Ferrante et al. (2013) used a set of manually pre-defined behaviors and conditions. The authors then used grammatical evolution to assemble these behaviors into a rule set controlling the robots in a foraging mission. The design methods of the AutoMoDe family follow a similar approach. Pre-defined modules—either manually (Francesca et al., 2014, 2015; Hasselmann and Birattari, 2020; Garzón Ramos and Birattari, 2020) or automatically (Ligot et al., 2020a; Hasselmann et al., 2023) crafted before design time—are assembled into higher-level control architectures, such as finite-state machines (Francesca et al., 2014, 2015; Cambier and Ferrante, 2022) or behavior trees (Kuckling et al., 2018b; Ligot et al., 2020b; Kuckling et al., 2021b). Duarte et al. (2015) preprogrammed several behavior primitives that were then combined into hierarchical control structures. If the modules are crafted specifically for the mission at hand (e.g., GESwarm), then the design process can only design control software for the considered mission. Furthermore, it should be classified as *semi-automatic* design, as the definition of the modules depends on the mission and thus becomes part of the design process. Conversely, if the modules are crafted in a mission-agnostic

way (e.g., AutoMoDe) then the design process can address a class of missions, implicitly defined by the set of modules. Furthermore, these design methods can be classified as either *automatic* or *semi-automatic* design, depending on whether further human intervention takes place.

Other approaches outside of swarm robotics could also fall into this level. In neural architecture search, an optimization algorithm searches for optimal topologies of deep neural networks (Elsken et al., 2019). These deep neural networks are composed of different layers, which can be combined and trained by the optimization algorithm.

6.3 Level 3 - Shaping

At Level 3, the design process can further shape its own architecture. This means, that almost all decisions are delayed and kept deliberately open for the design process. Notably, this includes not only the choice of modules and their parameters but also the architecture of the control software. Therefore, the optimization algorithm can choose between different architecture, such as neural networks or finite-state machines. Having chosen the architecture, the design process than chooses modules to assemble in the architecture. It is even possible that the design process can create its own modules and that these might be of different architectures as the whole instance of control software (for example, the design process might nest a neural network inside a finite-state machine).

The technique of shaping the architecture has been commonly employed in neuro-evolutionary swarm robotics. For example, Quinn et al. (2003) used an evolutionary algorithm to design control software in the form of artificial neural networks in a collective motion task. Each genome was an encoding of a randomly generated topology of an artificial neural network and the weights of its connections. During the design process, mutations could modify either the weights or the topology by adding or removing nodes from the network. Several other neuro-evolutionary design methods (Duarte et al., 2016; Hasselmann et al., 2021) used NEAT (Stanley and Miikkulainen, 2002) to design the weights and the topology of the neural networks and can therefore be classified as Level 3 optimization-based design methods. Unlike in neural architecture search, however, NEAT does not rely on pre-defined layers but has complete freedom in designing the topology of the artificial neural network. Similar to Level 1 neuro-evolutionary design methods, Level 3 neuro-evolutionary design methods usually address classes of missions rather than just a single one. Furthermore, they can be either *semi-automatic* or *automatic*, depending on the role of the human designer during the design process.

In the context of modular design, Hasan (2022) developed swarmDesign, an integrated framework that could design control software in the form of both

finite-state machines and behavior trees. The framework built on top of level 2 automatic design methods, such as AutoMoDe-`Chocolate`, AutoMoDe-`Maple`, and AutoMoDe-`TuttiFrutti`.

6.4 Outlook

The main challenge of optimization-based offline designs remains the reality gap. It is the main difference between the design challenge in swarm robotics and similar problems in related domains, such as supervised machine learning or reinforcement learning.

In the case of *optimization-based offline design for swarm robotics*, the solution to be found is an instance of control software. The instance of control software should result in an optimal behavior, given an environment (and the processes by which the robots interact within that environment). However, due to the limitations discussed in Sections 2.4 and 2.7, we do not have direct access to the target environment. Instead, the approach operates on a stand-in for the actual environment—a simulation environment. By the nature of simulations, these are only representations of the real environment and can never capture all the dynamics of the real world. Even realistic simulations can only ever be approximations of reality and will contain their own idiosyncrasies. This means that the design process operates on data that is generated by a different process (simulation) than the context in which the control software will be used (reality).

Related to optimization-based offline design of control software is the domain of *reinforcement learning*. In reinforcement learning, the interactions between the agents and the environment are assumed to be a Markov decision process (MDP) and a policy is learned that maximizes the reward obtained from the Markov decision process. However, as it is often assumed that the Markov decision process used for training is the process of interest, there often is little need for generalization. There have been some works to address the issue of generalization in the context of reinforcement learnig (Cobbe et al., 2019). The issue becomes similar prominent as for optimization-based offline design of control software, when policies are learned in simulation for embodied agents, e.g., robots (Muratore et al., 2022). In fact, we argue that the case of reinforcement learning in simulation for embodied agents is equivalent to optimization-based offline design of control software. However, reinforcement learning in the broader setting often lacks the generalization or transferability approach that is characteristic to optimization-based offline design of control software.

Another related domain is *supervised machine learning*. In supervised learning, the goal is to fit a model on a training set, generated by a process of interest, to predict the relationship between input and output. The model might be a classifier,

a regression model, or a recommender system. However, the interest of supervised learning is not on modelling the training set, but rather generalization (Goodfellow et al., 2016), that is, the performance of the model on previously unseen data. A major hypothesis in this approach is that the data for the training set is generated by the (same) process of interest that generates the unseen data for the application of the model. This implies that finding a predictor on the training set means finding a predictor for the process of interest. The training set is therefore a stand-in for the actual process of interest. The evaluation on the test set is necessary to ensure that the predictor did not over-fit the distribution of the training data (which due to sampling techniques and/or noise will vary from the "true" underlying distribution). Often, the training and test set are generated from the same sample of the process (e.g., from a train/test split). This comes at the risk that the sample itself was not representative of the process of interest. Consequently, another commonly employed technique is a validation set. The validation set, like the training and test set, is a sample obtained from the process of interest. However, it is sampled separately from the training and test set, to avoid introducing the same biases as in the training or test set. When the process of interest is time-invariant, this means that the hypothesis, that the training set is representative of the process of interest, remains unaffected. However, when the process of interest is not time-invariant this means that the training set might be "outdated" and not representative of the current state of the process.

While there are many similarities between reinforcement learning, supervised learning and optimization-based offline design of control software, the reality gap separates optimization-based offline design most clearly from the other domains. In particular, the presence of the reality gap is caused by the fact that in optimization-based offline design, the design process does not directly operate on the process of interest, whereas reinforcement learning and supervised learning assume that the learning process either operates directly on the process of interest or at least on data sampled from it. Therefore, addressing the reality gap needs to be a constant consideration, when developing optimization-based design methods for swarm robotics. We believe that any optimization-based offline design process needs to address the reality gap starting from the conception of the design process. Ultimately, this will lead to a trade-off between allowing and restricting the expressiveness of the control software.

Chapter 7

Conclusions & Future work

Optimization-based offline design of control software for robot swarms has shown to generate well-performing control software in a wide range of missions. Yet, despite its relevance, the study of the optimization algorithm has received little attention from the swarm robotics community. In this dissertation, I have presented my work on optimization in the automatic modular design of control software for robot swarms. It forms a first step towards understanding the role of optimization in the context of optimization-based offline design of control software for robot swarms. I have begun by surveying the literature in-depth and by concluding that only few works have studied more than one optimization algorithm. Among those few works (Francesca et al., 2015; Cambier and Ferrante, 2022) the optimization algorithms were a means to an end rather than the subject of study. I then argued that it would be necessary to study the impact of optimization, one of the central elements in optimization-based design.

To that end, I have created four design methods based on different optimization algorithms (`Cherry`, `IcePop`, `Schwarzwälder2`, and `Schwarzwälder3`). These design methods, together with `Chocolate`, cover local-search algorithms, model-free global search algorithms, and model-based global search algorithms. I have tested the developed design methods across several missions, often both in simulation and reality. The tools and protocols presented in this dissertation can be used in future work to study more aspects of the role of optimization.

I was able to make a few key observations that hold across all studies performed. The resilience of automatic modular design to the reality gap seems to be inherent to the modular design approach, and has not been affected much by the optimization algorithm. Indeed, all considered design methods were able to transfer satisfactorily onto real robots, whereas `EvoStick` usually failed. Furthermore, I observed that the selection of the optimization algorithm had an impact on the performance of the generated control software. For large design budgets, both local search algorithms (iterative improvement and simulated annealing) and a model-based optimization

algorithm (SMAC3) were able to outperform Iterated F-race. However, these gains in performance came at a price. In the case of the local search algorithms, it was necessary to specify the neighborhood function. The definition of this function implicitly defines the search space for the local search algorithms and an incorrect specification could hinder the ability of local search algorithms to find well-performing instances of control software. In the case of model-based optimization algorithms, I observed that the maintenance of the model resulted in a significantly increased run time of the design process.

Lastly, I identified three levels of optimization-based design. For each level, I described the conceptual approach to optimization that was encompassed in it and I discussed several past and future avenues to it.

7.1 Research contributions in detail

In Chapter 2, I have provided a review of the state of the art, with a special focus on optimization-based design of control software. In particular, I focussed on recent design methods of design approaches such as neuro-evolution, automatic modular design, or imitation learning. The results of the review highlighted that the role of the optimization algorithm has rarely been studied in the literature and thus motivated the research presented in this dissertation.

In Chapter 4, I have developed two automatic modular design methods based on local search algorithms. AutoMoDe-`Cherry` is uses iterative improvement as local search algorithm and AutoMoDe-`IcePop` uses simulated annealing. The results of this chapter provided new insights into the structure of the search space. In particular, it appeared that the search space is structured in such a way that local search-based design methods can easily design control software with similar performance as Iterated F-race. Furthermore, I studied a few selected hyperparameter combinations for the design methods. The results showed that no studied hyperparameter combination performed reliably better than the chosen default parameters.

In Chapter 5, I have focussed on model-based optimization algorithms. To that end, I have developed `Schwarzwälder2` and `Schwarzwälder3`, two automatic modular design methods that are based on SMAC2 and SMAC3, respectively. I have compared these design methods with `Chocolate` and `EvoStick` on a set of 30 randomly generated missions. The results of this chapter showed that `Schwarzwälder3` was able to find better fine-tuned software than `Chocolate` but in general the two design methods relied on similar strategies. Furthermore, the relative performance ranks between modular design methods encountered in simulation did not change when assessing the control software in reality.

Lastly, in Chapter 6, I developed my vision on optimization-based design of

control software for robot swarms. I identified three levels of optimization-based design and I have analyzed the characteristics of each of them. Additionally, I have discussed similarities and differences between these levels and related domains.

7.2 Future work

The research presented in this dissertation has opened some future research questions. For example, my research on local search algorithms showed that the chosen neighborhood structure seems to facilitate the design of well-performing instances of control software. An avenue for future research would therefore be to better understand the shape of the search space and its relation to the neighborhood function. Similarly, it would be of interest to investigate the effect of different representations of the search space used in the other design methods, such as `Chocolate`, `Schwarzwälder2`, and `Schwarzwälder3`.

Another interesting avenue of future work would be to better understand the implications of model-based optimization algorithms. In particular, the apparent time and performance trade-off between `Chocolate` and `Schwarzwälder3`, but also the reasons for which `Schwarzwälder3` outperformed `Schwarzwälder2` in simulation and reality.

Furthermore, some limitations to the work presented in this dissertation remain. For example, I only considered different optimization algorithms in missions that I knew could already be solved by Iterated F-race. Therefore, I did not investigate the limitations of each optimization algorithm in terms of the missions for which it can successfully design control software. It is likely, for example, that design methods based on local search algorithms will not be able to find well-performing instances of control software in mission with many local optima in their search space. For future work, it would be necessary to study how the different optimization algorithms perform in more complex missions.

Another avenue for future work is that I only considered to use the same optimization algorithm for the complete design process. Future work could therefore focus on combining different optimization algorithms within the same design method. For example, the design method could first run a design only on the modules (without parameter tuning) using a global search algorithm. In order to assess the performance of any selection of modules, a smaller design is run to tune the parameters of the selection, for example using local search. Alternatively, one might run a design with only rough parameter thresholds (e.g., high, medium, low) and the exact parameters are determined during an online tuning phase on the robots.

Appendix A

Behavior trees as an alternative control architecture

Besides the optimization algorithms, I also studied an alternative control architecture—behavior trees. Behavior trees have been adopted in several AI and robotics applications (Colledanchise and Ögren, 2018). Compared to finite-state machines, they offer increased modularity and human understandability. Additionally, they implement two-way control transfers. These properties make behavior trees an interesting alternative to finite-state machines.

This appendix is organized as follows. In Section A.1, I give a brief introduction into behavior trees. In Section A.2, I describe AutoMoDe-`Maple`, an automatic modular design method that assembles the modules of `Chocolate` into behavior trees. In Section A.3, I describe my experiments to design behavior trees using iterative improvement. In Section A.4, I present AutoMoDe-`Cedrata`, an automatic modular design method that uses modules that were specially designed to be used in behavior trees.

A.1 Behavior trees

In this section, I give a brief description of behavior trees and their functioning. I adopted the framework that Marzinotto et al. (2014) proposed to unify the different variants of behavior trees described in the literature. The original idea of behavior trees was proposed for the Halo 2 video game (Isla, 2005). Since then, behavior trees have found applications in many computer games, for example, Spore and Bioshock (Champandard et al., 2010). Recently, behavior trees have attracted the interest of the research community. Initial research focused on the automatic generation of behaviors in video games, for example, the commercial game DEFCON (Lim et al., 2010) and the Mario AI competition (Perez et al.,

2011). Even more recently, behavior trees have found applications in the control of unmanned aerial vehicles (Ögren, 2012), surgical robots (Hu et al., 2015), and collaborative robots (Paxton et al., 2017).

A behavior tree is a control architecture that can be expressed as a directed acyclic graph with a single root. With a fixed frequency, the root generates a *tick* that controls the execution. The tick is propagated through the tree and activates each node that it visits. The path that the tick takes through the tree is determined by the inner nodes, which are called control-flow nodes. Once the tick reaches a leaf node, a condition is evaluated or an action is performed. Then, the leaf node immediately returns the tick to its parent together with one of the following three values: *success*, *failure*, or *running*. A condition node returns *success*, if its associated condition is fulfilled; *failure*, otherwise. An action node performs a single control step of its associated action and returns *success*, if the action is completed; *failure*, if the action failed; *running*, if the action is still in progress. When a control-flow node receives a return value from a child, it either immediately returns this value to its parent, or it continues propagating the tick to the remaining children. There are six types of control-flow nodes:

Sequence ($\rightarrow$): ticks its children sequentially, starting from the leftmost child, as long as they return *success*. Because it does not remember the last child that returned *running*, it is said to be memory-less. Once a child returns *running* or *failure*, the sequence node immediately passes the returned value, together with the tick, to its parent. If all children return *success*, the node also returns *success*.

Selector ($?$): memory-less node that ticks its children sequentially, starting from the leftmost child, as long as they return *failure*. Once a child returns *running* or *success*, the selector node immediately passes the returned value, together with the tick, to its parent. If all children return *failure*, the node also returns *failure*.

Sequence* ($\rightarrow^*$): version of the sequence node with memory: resumes ticking from the last child that returned *running*, if any.

Selector* ($?^*$): version of the selector node with memory: resumes ticking from the last child that returned *running*, if any.

Parallel ($\rightrightarrows$): ticks all its children simultaneously. It returns *success* if a defined fraction of its children return *success*; *failure* if the fraction of children return *failure*; *running* otherwise.

Decorator (δ): is limited to a single child. It can alter the number of ticks passed to the child and the return value according to a custom function defined at design time.

Behavior trees offer a number of benefits over finite-state machines. In the context of automatic modular design, the most important properties of behavior trees are their enhanced expressiveness, the principle of two-way control transfers, and their inherent modularity (Ögren, 2012; Colledanchise and Ögren, 2018).

Ögren and coworkers have shown that behavior trees generalize finite-state

machines only with selector and sequence nodes (Ögren, 2012; Marzinotto et al., 2014). With parallel nodes, behavior trees are able to express individual behaviors that have no representation in classical finite-state machines. Additionally, behavior trees implement the principle of two-way control transfers: the control can be passed from a node to its child, and can also be returned from the child, along with information about the state of the system. In finite-state machines, the control flow is one-directional: the control goes from one state to another via a transition and cannot be returned. I foresee that this feature can be exploited to automatically design control software that is more robust to unexpected sensory input and that can naturally deal with failing behaviors. Finally, behavior trees are inherently modular: each subtree is a valid behavior tree. Due to this property, behavior trees are more easily manipulated than finite-state machines. Indeed, one can move, modify, or prune subtrees without compromising the structural integrity of the behavior tree. The modularity of behavior trees could simplify the conception of tailored optimization algorithm based on local manipulations.

A.2 AutoMoDe-`Maple`

Together with my coworkers (Kuckling et al., 2018b; Ligot et al., 2020b), I developed AutoMoDe-`Maple`. `Maple` is an automatic modular design method that generates control software in the form of behavior trees. It produces control software for the e-puck robot (see Chapter 3.2), with capabilities formalized through RM1.1 (see Chapter 3.3). The behavior trees are designed by selecting, combining, and fine-tuning a set of predefined modules: the six low-level behaviors and the six conditions defined by Francesca et al. (2014) for `Vanilla`, and later used in `Chocolate` (Francesca et al., 2015) (see Chapter 3.4.1). `Maple` was introduced with the purpose of exploring the use of behavior trees as a control architecture in the automatic modular design of robot swarms. To conduct a meaningful study of the potential of behavior trees as a control architecture, we compared `Maple` with `Chocolate`, an automatic modular design method that generates control software in the form of probabilistic finite-state machines (see Chapter 3.4). We conceived `Maple` to be as similar as possible to `Chocolate` so that differences in performance between the two methods can only be attributed to the different control architecture they adopt. `Maple` and `Chocolate` generate control software for the same robotic platform, they have at their disposal the same set of modules, and they use the same optimization algorithm, namely Iterated F-race (see Chapter 3.1).

In a probabilistic finite-state machine generated by `Chocolate`, every state has an associated low-level behavior and each transition has an associated condition. Because low-level behaviors (the states of the finite-state machine) are executed

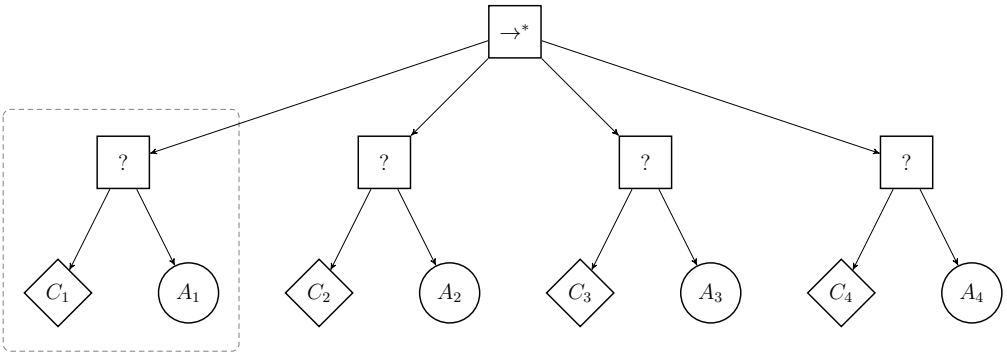


Figure originally published in Ligot et al. (2020b).

Figure A.1: Illustration of a behavior tree with restricted structure that `Maple` can produce. `Maple` generates a behavior tree by defining first the number of selector subtrees (highlighted by the dashed box), and by then specifying and fine-tuning the condition and action nodes that compose each selector subtree.

until an external condition (a transition) is enabled, they do not have inherent termination criteria. The absence of termination criteria implies that, when used as action nodes in a behavior tree generated by `Maple`, the low-level behaviors of `Chocolate` can only return *running*. As a result, part of the control-flow nodes of behavior trees do not work as intended—see Section A.1. If the goal would have been to develop a high-performing design method based on behavior trees, we would have needed to redefine the low-level behaviors so as to add the possibility to return *failure* and *success* when used as action nodes. However, modifying the low-level behaviors would have resulted in having a set of modules in `Maple` that is different from the one of `Chocolate`, which would not have allowed us to isolate the element we wish to study: the control architecture. Instead, `Maple` used the unmodified modules of `Chocolate`, and generated behavior trees with a restricted structure that only uses a subset of the control-flow nodes.

This restricted structure only used a subset of the control-flow nodes of the classical implementation of behavior trees. The top-level node was a sequence* node ($\rightarrow^*$) and could have up to four selector subtrees attached to it. A selector subtree was composed of a selector node ($?$) with two leaf nodes: a condition node as the left leaf node, and an action node as the right leaf node. Figure A.1 illustrates a behavior tree with the restricted structure adopted here. The maximal number of subtrees, and therefore the number of action nodes, was limited to four to mimic the restrictions of `Chocolate`, which generates probabilistic finite-state machines with up to four states.

In the example of Figure A.1, the left-most selector subtree (highlighted by the

dashed box) is first ticked and action A_1 is executed as long as condition C_1 returns *failure*. If condition C_1 returns *success*, the top-level node ($\rightarrow^*$) ticks the second selector subtree, and A_2 is executed, provided that C_2 returns *failure*. Because the top-level node is a control-flow node with memory, the tick will resume at the second subtree in the following control cycle. A_2 is therefore executed as long as C_2 returns *failure*. Although actions A_1 and A_4 are not in adjacent sub-trees, A_4 can be executed directly after A_1 granted that conditions C_1, C_2, and C_3 return *success* and C_4 returns *failure*. When condition C_4 of the last selector subtree returns *success*, the top-level node of the tree also returns *success* and no action is performed. In this case, the tree is ticked again at the next control cycle, and the top-level node ticks the left-most selector subtree again.

The size of the space spanning all possible instance of control software that can be produced by `Maple` is in $O\left(|\mathcal{B}|^4\,|\mathcal{C}|^4\right)$, where $\mathcal{B}$ and $\mathcal{C}$ are the sets of low-level behaviors and conditions, respectively (Kuckling et al., 2018a). The search space can be formally defined as:

$$\left[T, \#N^{(2)}, N_i^{(2)}, \#L_i, L_{ij}, L_{ij}^p\right], \quad \text{with } i = \left\{1, ..., \#N^{(2)}\right\}, j = \left\{1, ..., \#L_i\right\},$$

where $T \in \{\text{sequence*}\}$ is the type of the top-level node; $\#N^{(2)} \in \{1, ..., 4\}$ is the number of level 2 nodes; $N_i^{(2)} \in \{\text{selector}\}$ is the type of the level 2 node i; $\#L_i \in \{2\}$ is the number of leafs of node i; L_{ij} is the type of the j-th leaf of node i, with $L_{i1} \in \mathcal{C}$ and $L_{i2} \in \mathcal{B}$; and L_{ij}^p are the parameters of leaf L_{ij}.

A.2.1 Experiments

We studied `Maple` on two missions: FORAGING and AGGREGATION. In order to better appraise the performance of `Maple`, we also designed control software using `Chocolate`. The two missions were performed in a dodecagonal arena delimited by walls and covering an area of $4.91\,\mathrm{m}^2$. Twenty e-puck robots were distributed uniformly in the arena at the beginning of each experimental run, and the duration of the missions was limited to $120\,\mathrm{s}$.

FORAGING

Because the robots cannot physically carry objects, an idealized form of foraging was considered. In this version, an item was considered picked up when a robot entered a source of food, and that a robot dropped a carried item when it entered the nest. A robot could only carry one item at a time. In the arena, a source of food was represented by a black circle, and the nest was represented by the white area (see Figure A.2). The two black circles had a radius of $0.15\,\mathrm{m}$, they were separated by a distance of $1.2\,\mathrm{m}$, and were located at $0.45\,\mathrm{m}$ away from the white area. A

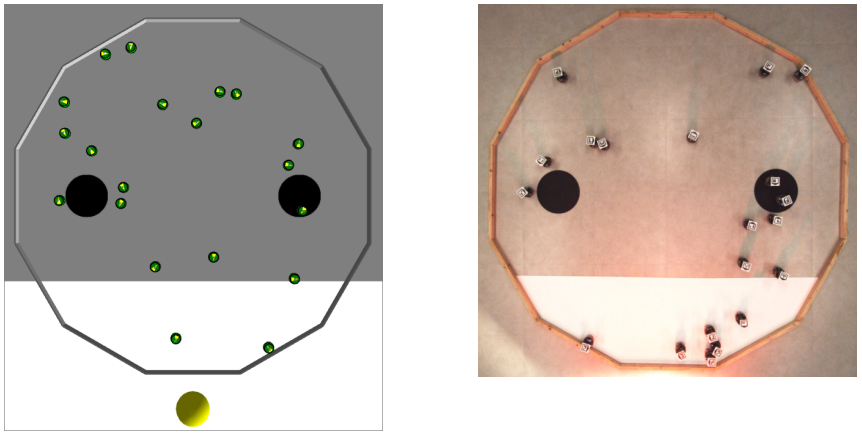


Figure originally published in Ligot et al. (2020b).

Figure A.2: FORAGING. *Left*: Simulated arena. *Right*: Real arena. The red glow visible in the picture is due to a red gel we placed in front of the light source. With the red gel, the light does not disturb the overhead camera that is used to track the position of the robots and compute the objective function. Yet, the light is still perceived by the robots that use their infrared sensors to sense it.

light source was placed behind the white area to indicate the position of the nest to the robots.

The goal of the swarm was to retrieve as many items as possible from the sources to the nest. The objective function was defined as:

$$F_{For} = I, \tag{A.1}$$

where I is the number of items deposited in the nest.

AGGREGATION

The swarm had to select and aggregate on one of the two black areas (see Figure A.3). The two black areas had a radius of $0.3\,\mathrm{m}$ and were separated by a distance of $0.4\,\mathrm{m}$. The objective function was defined as:

$$F_{Agg} = max(N_l, N_r)/N, \tag{A.2}$$

where N_l and N_r are the number of robots located on the left and right black area, respectively; and N is the total number of robot in the swarm. The objective function was computed at the end of a run and was maximized when all the robots had aggregated on the same black area.

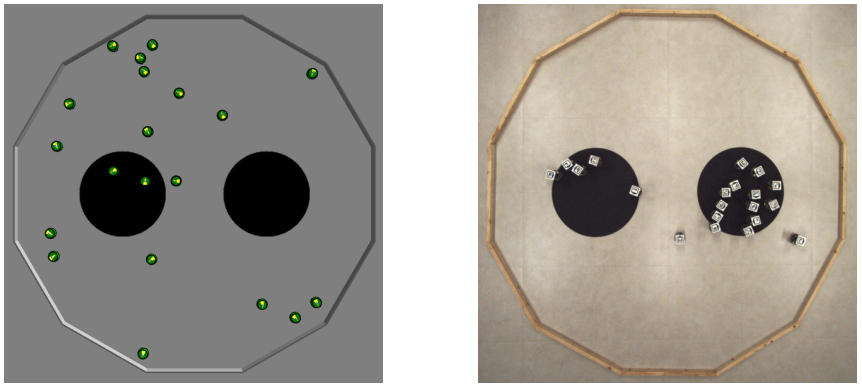


Figure originally published in Ligot et al. (2020b).

Figure A.3: AGGREGATION. The objective function F_{A} is computed as the maximal fraction of robots situated either on the left area (N_l/N) or on the right area (N_r/N). It is evaluated at the end of an experimental run. *Left*: Simulated arena, with $F_{\text{A}} = 0.1$ as 2 robots stand on the left black area ($N_l = 2$) and no robot stands on the right one ($N_r = 0$). *Right*: Real arena, with $F_{\text{A}} = 0.65$ as $N_l = 5$ and $N_r = 13$.

Dummy control software

Throughout the three studies, we compared the performance of automatically generated control software to the one of two instances of control software—one per mission—that we called "dummy" control software. They performed a simple, naive and trivial behavior that we considered as a baseline for each mission. With this comparison, we assessed whether the automatic design methods could produce behaviors that were more sophisticated than trivial solutions. To produce the two instances of dummy control software, we used the same low-level behaviors and conditions that `Maple` and `Chocolate` had at their disposal to generate control software, see Section 3.4.1. For FORAGING, we considered a strategy in which the robots moved randomly in the environment. We obtained this strategy by using the low-level behavior Exploration. For AGGREGATION, we considered a strategy in which the robots explored the environment randomly, and stopped when they encountered a black spot. We obtained this strategy by combining the modules Exploration, Black Floor, and Stop. To fine-tune the parameters of the modules, we used Iterated F-race with a design budget of 1000 simulation runs.

Protocol

We considered a swarm of 20 e-puck robots. To account for the stochasticity of the design process, we executed each design method several times, and therefore

produce several instances of control software. The number of executions of the design methods varied with the study. To evaluate the performance of a design method, each instance of control software was executed once in simulation. In Study 1, each instance of control software was also executed once in reality.

Simulations were performed with ARGoS3 and real robot experiments were performed in the IRIDIA experimental arena (see Section 3.6). During an evaluation run, some robots tipped over due to collisions. To avoid damages, we intervened to put them upright.

The performance of the design methods in the three studies is presented in the form of box-and-whiskers boxplots (see Section 3.7). In addition, the median performance of the dummy control software assessed in simulation is presented with a dotted horizontal line. The instances of control software produced, the experimental data collected in simulation and in reality, and videos of the behavior displayed by the swarm of physical robots are available online as supplementary material (Kuckling, 2023).

A.2.2 Results

Performance in simulation and reality

In the first study, `Maple`'s ability to produce control software that crosses the reality gap satisfactorily was evaluated. To do so, we compared the performance of control software generated by three design methods—`Maple`, `Chocolate` and `EvoStick`—both in simulation and in reality. Previous research (Francesca et al., 2015) indicated that `Chocolate` crosses the reality gap more satisfactorily than `EvoStick`. Francesca et al. (2014, 2015) argued that `Chocolate`'s ability to cross the reality gap is mainly due to its modular nature. Because `Maple` shares with `Chocolate` the same modular nature and differs from it only in the control architecture adopted, we expected `Maple` to also experience smaller performance drops than `EvoStick`.

We executed each design method 10 times, and thus produced 10 instances of control software. The design budget allocated to each method was 50k simulation runs. The results are depicted in Figure A.4.

FORAGING. In simulation, the performance of the control software produced by the three automatic design methods was similar, and was significantly better than the one of the dummy strategy. In reality, `EvoStick` was significantly worse than `Maple` and `Chocolate`. The performance of all three methods dropped significantly when passing from simulation to reality, but `EvoStick` suffered from the effects of the reality gap the most.

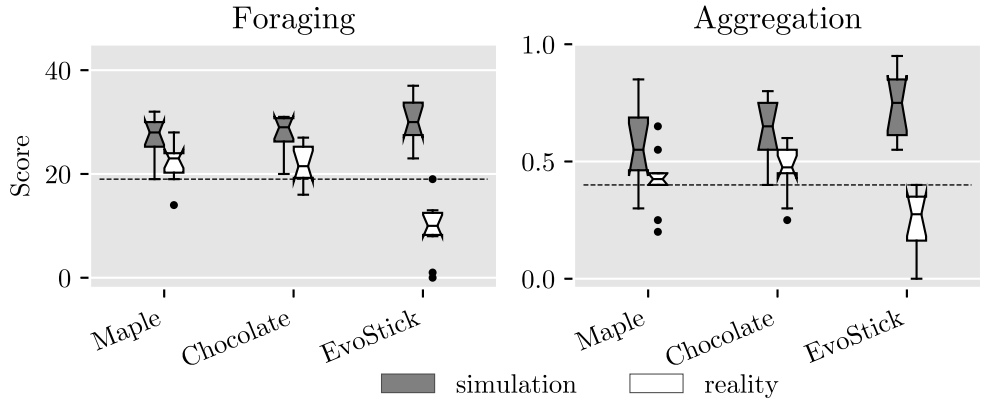


Figure adapted from Ligot et al. (2020b).

Figure A.4: Results of Study 1. The dark grey boxes represent performance obtained in the design context, white boxes represent performance obtained in reality. The dotted line represents the median performance of the dummy control software assessed in simulation (see Section A.2.1).

Most of the instances of control software generated by `Maple` and `Chocolate` displayed similar strategies: the robots explored the environment randomly and once a black area (that is, a source of food) was found, they navigated towards the light to go back to the white area (that is, the nest). One instance of control software produced by `Maple` used the Anti-Phototaxis low-level behavior to leave the nest faster once an item has been dropped. Three instances of control software produced by `Chocolate` displayed an even more sophisticated strategy: the robots only explored the gray area in the search for the sources of food. In other words, the robots always directly left the nest if they entered it, independently of whether they dropped an item or not. The performance drops experienced by `Maple` and `Chocolate` when porting the instances of control software from simulation to reality were probably due to the fact that, sometimes, robots are unable to move due to frictions. When this happened, they failed to contribute to the foraging process.

In simulation, the instances of control software generated by `EvoStick` displayed fundamentally different behaviors than the ones produced by `Maple` and `Chocolate`: the robots navigated following circular trajectories that crossed at least one source of food and the nest. In reality, the robots were not able to reproduce these circular trajectories. Contrarily to `Maple` and `Chocolate`, and with the exception of few cases, the instances of control software generated by `EvoStick` did not display an effective foraging behavior.

AGGREGATION. In simulation, EvoStick performed significantly better than Maple and Chocolate, which showed similar performance. In reality, a rank inversion was observed: Maple and Chocolate performed significantly better than EvoStick. Indeed, the performance of EvoStick dropped considerably, whereas the performance drop experienced by Maple and Chocolate was smaller.

The instances of control software produced by Maple and Chocolate efficiently searched the arena and made the robots stop on the black areas once they are found. In simulation, with the control software produced by EvoStick, the robots followed the border of the arena and then adjusted their trajectory to converge towards neighboring peers that were already situated on a black spot. In reality, the control software generated by EvoStick did not display the same behavior: robots were unable to find the black areas as efficiently as in simulation because they tended to stay close to the borders of the arena. Moreover, the robots tended to leave the black areas quickly when they were found. Although the three design methods performed significantly better than the dummy control software in simulation, none of the methods produced control software that made the physical robots reach a consensus on the black area on which they should aggregate.

Influence of the design budget

In the second study, the performance of Maple and Chocolate across different design budgets was investigated. Because the search space (that is, all instances of control software that can be generated) of Chocolate is significantly larger than the one of Maple—$O\left(|\mathcal{B}|^4|\mathcal{C}|^{16}\right)$ and $O\left(|\mathcal{B}|^4|\mathcal{C}|^4\right)$, respectively (Kuckling et al., 2018a)—we expected Maple to converge to high performing solutions faster than Chocolate.

Six design budgets were considered: 500, 1000, 5000, $10\,000$, $50\,000$ and $200\,000$ simulation runs. For each design budget, we executed each design method 20 times, and thus produced 20 instances of control software. In total, the two design methods were executed 120 times each. The results are depicted in Figure A.5.

FORAGING. The performance of the methods showed different trends when the design budget increased. For Maple, there was a significant improvement of the performance between design budgets of 1k and 5k, and between 50k and 200k simulation runs. For Chocolate, the performance increased significantly between design budgets of 5k and 10k, 10k and 50k, and 50k and 200k simulation runs.

With very small design budgets—0.5k and 1k simulation runs—Maple and Chocolate showed similar performance: they were unable to find solutions that

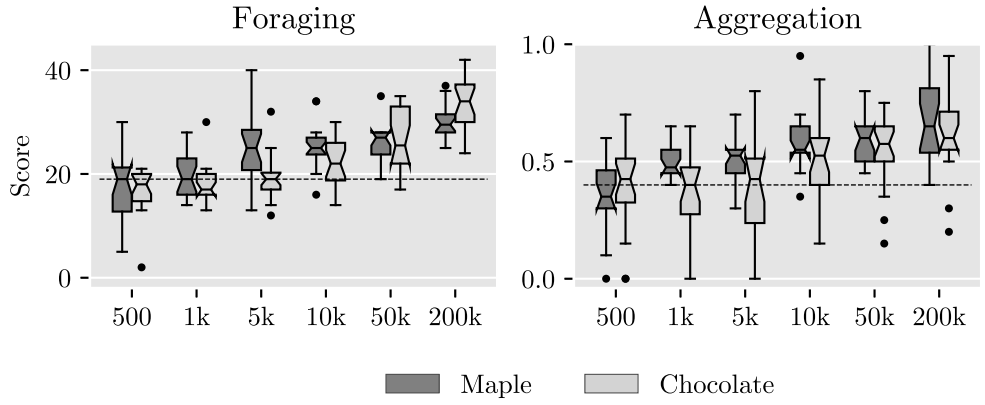


Figure adapted from Ligot et al. (2020b).

Figure A.5: Performance of `Maple` and `Chocolate` over multiple design budgets, expressed in number of simulation runs. The dotted line represents the median performance of the dummy control software (see Section A.2.1).

were better than the dummy control software. With a small design budget—5k simulation runs—`Maple` performed significantly better than `Chocolate`. Also, with 5k simulation runs, `Chocolate` and the dummy control software showed similar performance. With a large design budget—200k runs—`Chocolate` performed significantly better than `Maple`. Indeed, the instances of control software generated by `Chocolate` displayed a more sophisticated foraging strategy than those generated by `Maple`: to increase the rate of discovery of the food sources, the robots only explored the gray area of the arena, and stayed away from the nest. It appears that, with `Maple`'s restrictions on the structure of the behavior trees, it is impossible to produce the same strategy. Rather, with `Maple`, the robots explored the whole arena in order to find the food sources. Finally, the behavior trees generated by `Maple` with a design budget of 5k simulation runs were only outperformed by probabilistic finite-state machines when 200k simulation runs were allocated to `Chocolate`.

AGGREGATION. The performance of the control software generated by both methods increased almost constantly with the design budget. Also for this mission, `Chocolate` required a design budget of at least 10k simulation runs in order to generate control software that was significantly better than the dummy control software. Contrarily, `Maple` only required 1k simulation runs. With 1k and 5k simulation runs, `Maple` outperformed `Chocolate`. For larger design budgets, `Maple` and `Chocolate` showed similar performance.

Although the design budgets considered allowed the two methods to outperform

the dummy control software in multiple occasions, neither of them generated control software that completed the mission satisfactorily. Indeed, the maximal median performance obtained was $F_{Agg} = 0.65$, which means that only 13 out of the 20 robots were on the same black spot.

`Maple` and some of its possible variants

In the third study, the changes in performance were explored when variations to the control architecture of `Maple` are introduced. The exploration of alternative architectures was not exhaustive: we only considered variants that generated behavior trees whose structure is similar to the one of the behavior trees generated by `Maple`. We limited our exploration to variants that generate trees with: 3 levels (top-level, inner, and leaf nodes); up to 4 branches connected to the top-level node; and exactly 2 leaf nodes per branch. Because the action nodes of `Maple` can only return *running*, many of the variants following these contraints are unable to combine low-level behaviors into meaningful and elaborate individual behaviors.

Figure A.6 illustrates behavior trees for which the leaf nodes are kept unchanged, but the inner node types may vary with respect to those of `Maple`. To refer to these variants, we use the following notation: $\langle top \rangle$ ($\langle inner \rangle$), where $\langle top \rangle$ and $\langle inner \rangle$ are the types of the top-level node and the inner nodes, respectively. With this notation, the class of behavior trees produced by `Maple` can be formally indicated as $\rightarrow^*(?)$—the top-level node is a sequence* and the inner nodes are selector nodes. If the inner nodes of a variant can be chosen from 2 different types, we refer to it as $\langle top \rangle$ ($\langle inner_1 \rangle | \langle inner_2 \rangle$). For example, each inner node of the variant $\rightarrow^*(\rightarrow|?)$ can be either a sequence or a selector node. We considered the following variants:

- $\rightarrow^*(\rightarrow)$: The top-level node is a sequence* node and the inner nodes are sequence nodes. See Figure A.6a. This variant is not interesting as only action A_1 can be executed, granted that condition C_1 returns *success*. If condition C_1 returns *failure*, no action is executed.

- $?^*(?)$: The top-level node is a selector* node and the inner nodes are selector nodes. See Figure A.6b. This variant is not interesting as only action A_1 can be executed, granted that condition C_1 returns *failure*. If condition C_1 returns *success*, no action is executed.

- $\rightarrow(?)$: The top-level node is a sequence node and the inner nodes are selector nodes. See Figure A.6c. This variant is similar to $\rightarrow^*(?)$ of `Maple`. However, because the top level node does not remember which subtree last returned *running*, the history of the past events is lost. In order for an action to be executed, all conditions situated to its left need to return *success*. Due to the

absence of memory, we do not consider this variant promising—and neither any other memory-less one.

$?^*(\rightarrow)$: The top-level node is a selector* and the inner nodes are sequence nodes. See Figure A.6d. In this variant, the action node of a subtree is executed as long as the condition returns *success*, whereas it is executed until the condition returns *success* in `Maple`. We included this variant in our study and we reported the performance of it under the name `CFN` (control-flow nodes).

$\rightarrow^*(\rightarrow|?)$: The top-level node is a sequence* and the inner nodes can be sequence or selector nodes. See Figure A.6e. This variant can produce meaningful behavior trees only if the inner nodes are selector nodes, which is exactly like in `Maple`'s $\rightarrow^*(?)$. Indeed, if the inner node of a given subtree is a sequence and the condition returns *failure*, then the execution of the behavior tree terminates. At the following iteration, the leftmost branch of the top-level node is ticked. Therefore, branches situated at the right of a sequence node are not likely to be ticked.

$?^*(\rightarrow|?)$: The top-level node is a selector* and the inner nodes can be sequence or selector nodes. See Figure A.6f. This variant can produce meaningful behavior trees only if the inner nodes are selector nodes, which is exactly like variant $?^*(\rightarrow)$. Indeed, if the inner node of a given subtree is a selector and the condition returns *success*, then the execution of the behavior tree terminates. At the following iteration, the leftmost branch of the top-level node is ticked. Therefore, branches situated at the right of a selector node are not likely to be ticked.

We also considered variants in which the inner nodes are kept unchanged, but the leaves are modified with respect to those of `Maple`. An illustration of some of these variants can be found in Figure A.7.

`FL` (free leaves): Each leaf node is to be chosen between condition and action nodes. See Figure A.7a. Four pairs of leaf nodes are possible: condition–condition (see first branch), condition–action (which corresponds to the leaf pair imposed in `Maple`, see second branch), action–condition (see third branch), and action–action (see fourth branch). For each subtree, the optimization algorithm is free to chose any pair of leaf nodes. The variant can express disjunction of conditions: a branch following a condition–condition leaf pair is ticked if the first or the second condition is met. However, the variant introduces dead-end states: when an action on the left hand side of a leaf pair is ticked, the action is executed for the remaining of the simulation run. We included this variant in our study.

`CA|CC` (condition–action or condition–condition): The right-hand side leaf node can be a condition or an action node. Two pairs of leaf nodes are thus possible: condition–action, condition–condition. With respect to `FL`, this variant can also express disjunction of conditions, but does not allow for dead-end states. We included this variant in our study.

`ND` (negation decorator): A negation decorator node can be instantiated above a condition node. See Figure A.7b. The negation decorator returns *failure* (*success*) if the condition returns *success* (*failure*). With the set of conditions available, it is particularly interesting to place a negation decorator above a condition on the color of the ground perceived (that is Black Floor, Grey Floor, or White Floor). Indeed, placing a negation decorator node above a Neighborhood Count condition is equivalent to having an Inverted Neighborhood Count condition, and vice versa. Similarly, a negation decorator above a Fixed Probability condition with ρ is equivalent to a Fixed Probability with $1 - \rho$. However, a negation decorator above a condition on a given color is equivalent to assessing the conditions for the two other colors simultaneously. We included this variant in our study.

`SP` (success probability): This variant adopts `Maple`'s $\rightarrow^*(?)$, but each action node has a probability ρ to return *success*. The probability ρ is a real value in the range $[0, 1]$ and is tuned by the optimization process. With this probability, we simulate the capability of the action nodes to assess if the low-level behaviors are successfully executed. We included this variant in our study.

We assessed the performance of different variations of the control architecture, namely `CFN`, `FL`, `ND`, `SP` and `CA|CC`. For each variant, 20 instances of control software were produced, all generated by the same optimization process with a design budget of 50k simulation runs. We compared the performance of the variants to the one of `Maple`.

FORAGING. None of the variants outperformed `Maple`. The methods `Maple`, `ND`, and `CA|CC` performed similarly. Moreover, they outperformed `CFN`, `FL`, `SP`, and the dummy control software. The variants `CFN`, `FL`, and the dummy control software showed similar performance.

All the instances of control software generated by `Maple` showed similar behaviors: the robots explored the arena until they found one of the food sources, then navigated towards the nest using the light as a guidance. In some cases, the robots used the Anti-Phototaxis low-level behavior to directly leave the nest once they deposited an item.

With variant ND, we could manually design control software that displayed an elaborate strategy: the robots increased the rate at which they discover food sources by only exploring the gray area of the arena. This behavior cannot be expressed by Maple (see Section A.2.2). An example of a behavior tree adopting variant ND that display this strategy is illustrated in the supplementary material (Kuckling, 2023). In this example, the elaborate strategy only emerged if the success probability of the condition node below the negation decorator was set to 1. Indeed, if the success probability was slightly lower, the behavior displayed was radically different, and more importantly, inefficient. It appears that, with the allocated budget, this necessary condition made it unlikely for Iterated F-race to produce this strategy.

Iterated F-race was not able to take advantage of the disjunction of conditions that is available in CA|CC to find better solutions that those of Maple. Indeed, we were unable to do so ourself. However, the increased search space of CA|CC does not hinder the optimization process as results obtained are similar to those of Maple.

In variant SP, the success probabilities, together with the conditions, were termination mechanisms for the subtrees. The additional termination mechanisms made it harder for Iterated F-race to exploit correlations between conditions and actions that led to behaviors as efficient as those generated by Maple. Most of the produced control software relied essentially on the Exploration low-level behavior.

With variant CFN, one can generate a behavior tree that expresses the same elaborate strategies that can be generated with variant ND. However, CFN is faced with a similar problem as ND: the success probability of the conditions needs to be set to 1 in order for that elaborate strategy to emerge. With a success probability set to a lower value, the condition node might return *failure* even though its condition is met, and the subtree might therefore terminate prematurely. The allocated design budget was not large enough for Iterated F-race to find behavior trees with meaningful connections between the conditions and behaviors, which resulted in poor performance.

The performance of the variant FL showed the highest variance. Sometimes, the behavior trees generated were similar to those produced by Maple. However, in many cases, the left leaf node of subtrees was an action node with an associated Exploration low-level behavior. Once this node is reached, this low-level behavior was executed until the end of the experimental run. As a result, the performance observed was similar to the one of the dummy control software.

AGGREGATION. Variant CFN outperformed Maple. Maple, FL, ND, and SP showed similar performance. Maple outperformed CA|CC. Every variant produced behavior trees that outperformed the dummy control software.

All the instances of control software generated by Maple and the different vari-

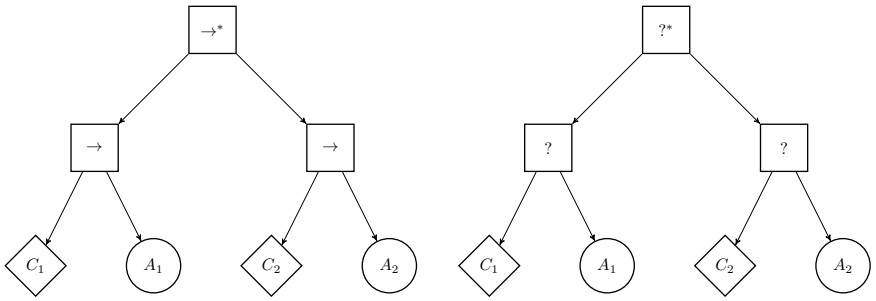
ants made use of the Exploration and Attraction low-level behaviors to efficiently search for the black spots. Maple and FL used the Stop low-level behavior for the robots to stay on the discovered spot. Contrarily, the majority of the behavior trees adopting variant CFN, ND, SP, and CA|CC did not have the Stop low-level behaviors as action nodes. Instead, they took advantage of the fact that, when no action node was executed, the robot stood still. CFN is the only variant for which Iterated F-race was able to exploit this feature to outperform Maple.

A.2.3 Discussion

We devised Maple to be as similar as possible to Chocolate: the two methods share the same optimization algorithm, the same set of predefined modules, and generate control software on the basis of the same reference model. The only difference between Maple and Chocolate is the control architecture adopted. The results show that Maple is robust to the reality gap. Indeed, Maple and Chocolate performed similarly, and they suffered from a reduced performance drop with respect to EvoStick, an evolutionary swarm robotics method. These results confirm Francesca et al. (2014) conjecture that AutoMoDe is robust to the reality gap due to its modular nature. They also indicate that the architecture into which the predefined modules are combined is a secondary issue.

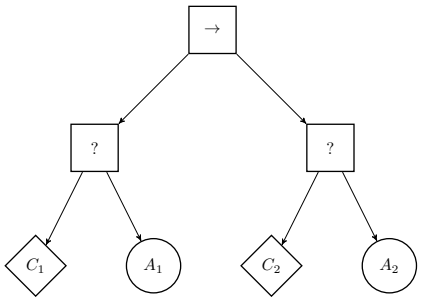
However, the restrictions on the structure of the behavior trees have shown that they inhibit Maple's expressiveness. Indeed, for FORAGING and with a large design budget, Chocolate was able to generate more efficient solutions that cannot be expressed by Maple. Conversely, Maple is able to converge to efficient solutions faster than Chocolate because of the smaller search space. The restrictions on the behavior trees produced by Maple, imposed by the absence of termination criteria of the low-level behaviors adopted, thus seem to be a double-edged sword: they facilitate the initial search for efficient solutions, but curb the expressiveness of behavior trees.

Future work should develop along two avenues. The first one should be dedicated to further investigate the use of Vanilla's and Chocolate's low-level behaviors as action nodes of behavior trees. For example, the control software generated by Maple with different design budgets should be assessed in robot experiments. The same holds for control software generated by Maple's variants. Also, further variants should be explored by relaxing the restrictions on the number of levels, branches, and leaves. For the relevant ones, the effect of the design budget should be investigated. As a second avenue, future work should be devoted to defining a high-performing design method based on behavior trees. To do so, one should first devise low-level behaviors with appropriate termination criteria—that is, behaviors that returns *success*, *failure*, or *running*. Then, one should develop an ad-hoc optimization algorithm that takes advantage of the inherent modularity of

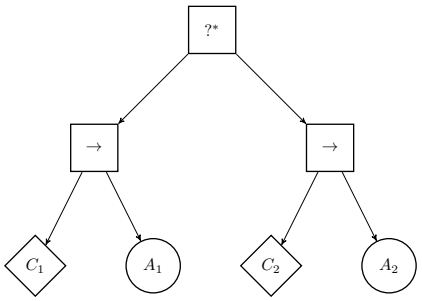


(a) Variant $\rightarrow^*(\rightarrow)$

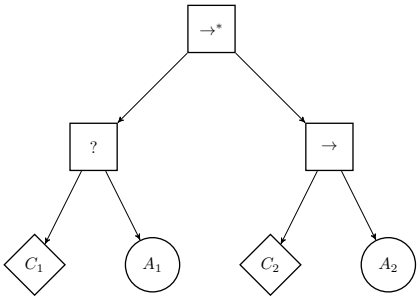
(b) Variant ?*(?)



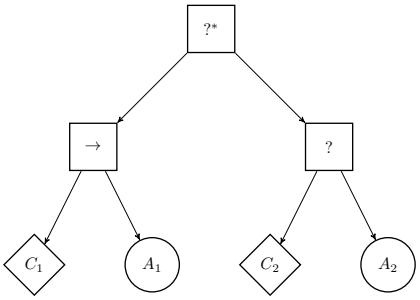
(c) Variant $\rightarrow(?)$



(d) Variant ?*($\rightarrow$) or `CFN`



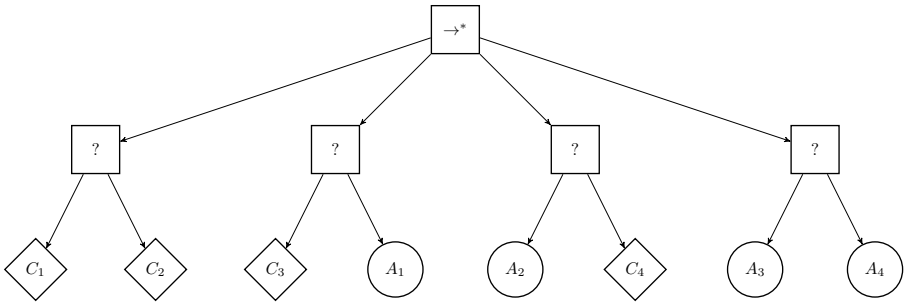
(e) Variant $\rightarrow^*(\rightarrow|?)$



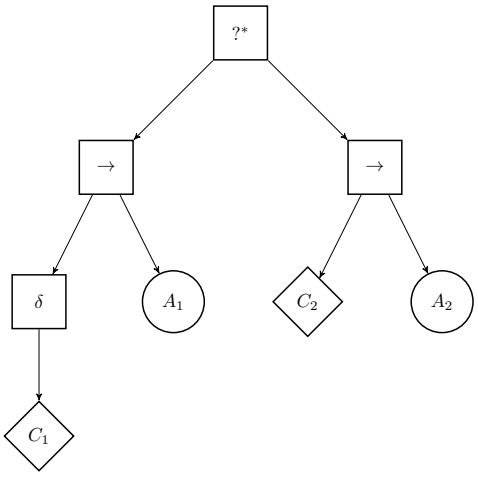
(f) Variant ?*($\rightarrow|?$)

Figure originally published in Ligot et al. (2020b).

Figure A.6: `Maple`'s variants. In these variants, the arrangement of the leaf nodes is unchanged with respect to `Maple`. The number of branches connected to the top-level node, and their order, has been determined without any a priori.



(a) Variant `FL` (free leaves)



(b) Variant `ND` (negation decorator)

Figure originally published in Ligot et al. (2020b).

Figure A.7: `Maple`'s variants. In these variants, the inner nodes are unchanged with respect to `Maple`. The number of branches connected to the top-level node, and their order, has been determined without any a priori.

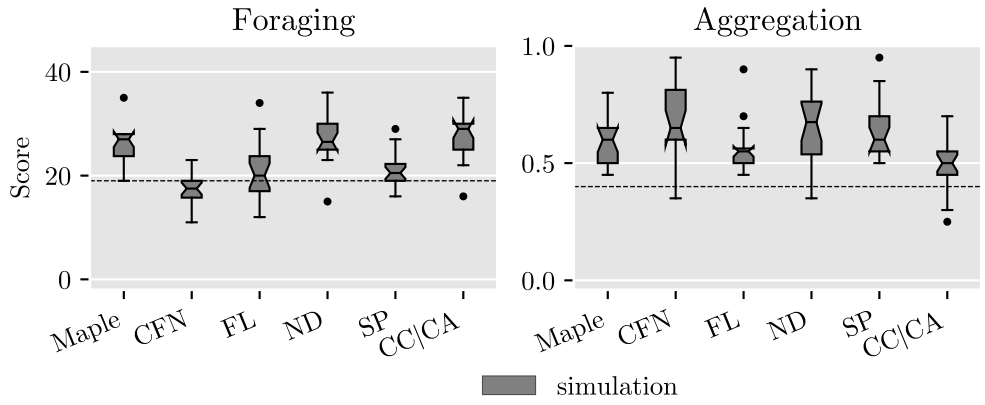


Figure adapted from Ligot et al. (2020b).

Figure A.8: Performance of different variants of `Maple`. The dotted line represents the median performance of the dummy control software (see Section A.2.1).

behavior trees.

A.3 AutoMoDe-Cherry-BT

I have also developed AutoMoDe-Cherry-BT, a variant of Cherry that assembles behavior trees instead of finite-state machines. The behavior trees follow the same constraints as Maple (see Section A.2). Consequently, I had to define a new neighborhood structure, instead of the one used for finite-state machines (see Section A.3.1). All other elements of Cherry-BT are the same as for Cherry.

A.3.1 Neighborhood structure

In the context of this work, I only considered behavior trees that followed a restricted architecture, as described for Maple. Behavior trees in the restricted architecture have three levels of nodes. The top-level node is a sequence* node. Underneath it are up to four so-called condition-action subtrees; composed of a selector node with a condition node as the first child and an action node as the second child. This architecture allows a behavior in an action node to be executed until the condition in its sibling condition node is met. From the following tick, the next condition-action subtree will receive the tick and execute the next behavior.

In this section, I describe additional properties that a behavior tree must have in order to be considered a valid instance of control software. I also describe a set of perturbation operators that transform a valid behavior tree into another valid one.

Validity

In the context of this work, a behavior tree is considered to be a valid instance of control software if it fulfills the following criteria:

V1) **Root node**: The behavior tree has exactly one root node.

V2) **Top-level node**: The behavior tree has exactly one top-level node (the sole child of the tick-generating root node). The top-level node is a sequence* ($\rightarrow^*$) node.

V3) **Number of subtrees**: The top-level node has between one and four children, all of which are selector (?) nodes.

V4) **Condition-action subtree**: Each selector (?) node has exactly two children. The first (left) child is a condition node and the second (right) child is an action node.

Perturbation operators

P1) **Add selector subtree**: If the behavior tree has less than the maximum number of selector subtrees, let i be the number of selector subtrees and $0 \leq j \leq i$ be an integer.
Add a selector subtree after the jth child of the top-level node ($j = 0$ meaning the new tree is added as the first child).

P2) **Remove selector subtree**: If the behavior tree has at least two selector subtrees, let t be a selector subtree.
Remove the selector subtree t from the tree.

P3) **Change subtree order**: If the behavior tree has at least two selector subtrees, let t be a selector subtree, i be the number of selector subtrees, j the position of t in the tree, and $1 \leq k \leq i$ be an integer, but $k \neq j$.
Move the selector subtree t to be the kth child of the top-level node.

P4) **Change condition of condition node**: Let c be a condition node.
Change the associated condition of c to a different one from the set of modules.

P5) **Change behavior of action node**: Let a be an action node.
Change the associated behavior of a to a different behavior from the set of modules.

P6) **Change parameter of a condition**: Let c be a condition node and p a parameter of the associated condition of c.
Set a new value for p within the bounds defined for the condition.

P7) **Change parameter of a behavior**: Let a be an action node and p a parameter of the associated behavior of a.
Set a new value for p within the bounds defined for the behavior.

Perturbation operators P1-P3 are structural perturbations, that is, they change the graph representation of the behavior tree. In our specific restricted architecture, P3 has no structural effect on the graph structure, as the condition-action subtrees are structurally identically, reordering them does not change the overall structure of the behavior tree. However, it has the potential to create structural changes in a less restricted setting, therefore it is justified to classify it as a structural perturbation as well. P4-P5 are modular perturbations and P6-P7 are parametric perturbations, as they only influence a single module or its parameters, respectively.

A.3.2 Experiments

I validated `Cherry`-BT in the same experimental setup as `Cherry` (see Section 4.3.1). In particular, I investigated three different cases for the initial instances of control software (`Cherry`-BT-`Minimal`, `Cherry`-BT-`Random`, `Cherry`-BT-`Hybrid`) and designed control software for the four missions AAC, SCA,

FORAGING, and GUIDED SHELTER.

A.3.3 Results

For the sake of completeness, I report again on the results obtained in the experiments described in Section 4.3 together with those performed for Cherry-BT and its variants. All results are available in the supplementary material (Kuckling, 2023).

Results 12.5k

Figure A.9 shows the performance of all design methods when they are allocated a budget of 12 500 (12.5k) simulation runs. In all missions except GUIDED SHELTER, the design methods Maple, Chocolate, Cherry-BT-Minimal, Cherry-Minimal, Cherry-BT-Random, Cherry-Random, Cherry-BT-Hybrid, Cherry-Hybrid performed similar to their counterpart that used the alternative architecture (behavior tree or finite-state machine). The only exceptions were Cherry-BT-Hybrid and Cherry-Hybrid. In the mission AAC, Cherry-Hybrid was significantly better than Cherry-BT-Hybrid, and in the mission FORAGING, Cherry-BT-Hybrid was significantly better than Cherry-Hybrid. The performance I registered for all design methods generating control software in the form of behavior trees was similar, except for Cherry-BT-Hybrid outperforming AutoMoDe-Maple in the mission GUIDED SHELTER.

In all four missions, EvoStick exhibited a large drop in performance, when assessed in pseudo-reality. The other design methods did not exhibit such a large drop, except in the mission SCA, where every design method except for Cherry-BT-GP, Maple and Cherry-BT-Hybrid suffered from a large performance drop when assessed in pseudo-reality. This is an indicator that these design methods might have a good transferability, allowing them to cross the reality gap satisfactorily.

Results 25k

Figure A.10 shows the results for all design runs with a budget of 25 000 (25k) simulations. In three of the four missions (AAC, FORAGING, and GUIDED SHELTER), Cherry-Minimal and Cherry-Random outperformed their respective counterparts Cherry-BT-Minimal and Cherry-BT-Random. Additionally, Cherry-Hybrid outperformed Cherry-BT-Hybrid in the missions AAC and GUIDED SHELTER. For the design methods Maple, Cherry-BT-Minimal, Cherry-BT-Random and Cherry-BT-Hybrid I registered similar performance throughout

all four missions, although in the mission FORAGING Cherry-BT-Random outperformed Maple and Cherry-BT-Minimal. EvoStick was able to generate sufficiently good control software in the design context, outperforming several other methods.

Comparison of the generated behavior trees in the mission FORAGING yielded no apparent different strategies, and as such the differences in performance can only be explained in such a way that Cherry-BT-Random was able to select more effective parameters than Maple or Cherry-BT-Minimal.

When assessed in a pseudo-reality context, however, EvoStick was the worst performing design method, and was outperformed by all other methods in the three missions AAC, FORAGING, and GUIDED SHELTER. In SCA, EvoStick performed best in the design context and although it suffered from a significant drop of performance when assessed in pseudo-reality, it still ranged as one of the best performing design methods in pseudo-reality.

In the mission SCA, the designed finite-state machines suffered from a larger performance drop when assessed in pseudo-reality. This could be an indicator that these instances of control software were overdesigned for the specific design context and would not transfer well into reality. Yet, the performance in pseudo-reality was still similar to the performance of the generated behavior trees in pseudo-reality.

Results 50k

Figure A.11 shows the results of all design methods for a budget of $50\,000$ (50k) simulations. In three of the four missions (AAC, FORAGING, and GUIDED SHELTER), the best design method for finite-state machines was able to outperform the best design method for behavior trees. This further supports my previous understanding that behavior trees, in the restricted topology, are too constrained and do not provide the same expressiveness as the finite-state machines (Kuckling et al., 2018b,a). In the missions AAC, SCA and FORAGING, I registered lower performance for those behavior trees designed by Cherry-BT-GP than those designed by the other methods. All three approaches to choosing the initial instance of control software—Minimal, Random, and Hybrid, performed similarly through all four missions (for a fixed architecture).

Throughout all four missions, EvoStick suffered from the largest pseudo-reality gap. The control software generated in the form of behavior trees showed only small drops in performance when assessed in pseudo-reality, as in the previous experiments. The finite-state machines designed by Chocolate also experienced small drops of performance, while the finite-state machines generated by iterative improvement, showed larger drops of performance when assessed in pseudo-reality. This could be an indicator of potential overdesign for these design methods,

however, the effect was not as big as for EvoStick.

A.3.4 Discussion

The design methods that operate on behavior trees did not improve their solution quality significantly when the budget was increased. This could be an indication that all considered design methods for the restricted behavior tree architecture found solutions close to the actual optimum for that restricted architecture. This reasoning seems to be supported by the fact, that neither Cherry-BT-GP nor Maple, two design methods based on optimization algorithms that are known to avoid local optima, found any solutions outperforming the iterative improvement based design methods. Unfortunately, the design methods that operate on the restricted behavior tree architecture failed to achieve the same performance as those that operated on finite-state machines. This can be attributed to the smaller search space, defined through the restricted architecture (Kuckling et al., 2018a). While the design process converged more quickly towards the suspected optimum, it could not reach the same solution quality as for the finite-state machines.

The investigation of the generated control software and the resulting behaviors has not shown any indication, as to why finite-state machines suffer more strongly from the pseudo-reality gap than the generated behavior trees. I can only speculate about possible reasons, but future work will be required to confirm or reject these hypotheses. One hypothesis would be that the bias/variance trade-off can also be observed through the restrictions applied to the behavior tree architecture. In the restricted behavior tree architecture, the execution of the current behavioral module can only be terminated by meeting a single condition. In the generated finite-state machines, many states have however at least two different outgoing transitions with different associated conditions. Therefore, when assessed in pseudo-reality, there are at least two different transitions, which could trigger prematurely or delay triggering, thus altering the behavior of a robot.

Across all missions, all design methods based on iterative improvement performed similarly. Differences that may be observed for a given mission or a given budget did not generalize to all observed results. This indicates that, also for the behavior trees, for any considered combination of budget and mission, the starting solution did not seem to play an important role in the solution quality.

A.4 AutoMoDe-Cedrata

As highlighted in our study on Maple, the lack of behavioral modules to return *success* and *failure* required us to impose a rigid architecture that could only express linear sequences of behaviors. In order to overcome this limitation,

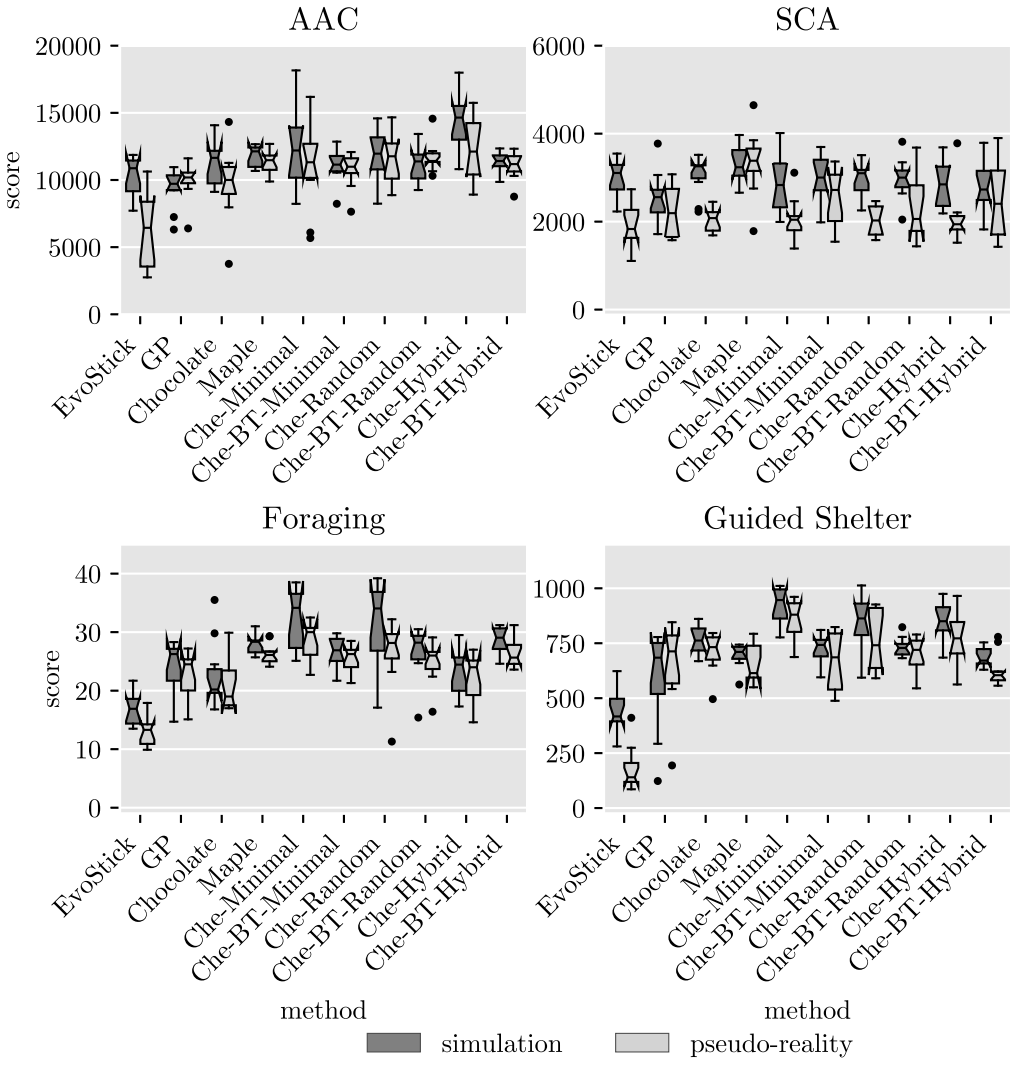


Figure adapted from Kuckling et al. (2020a).

Figure A.9: Results for all design methods for a budget of 12.5k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

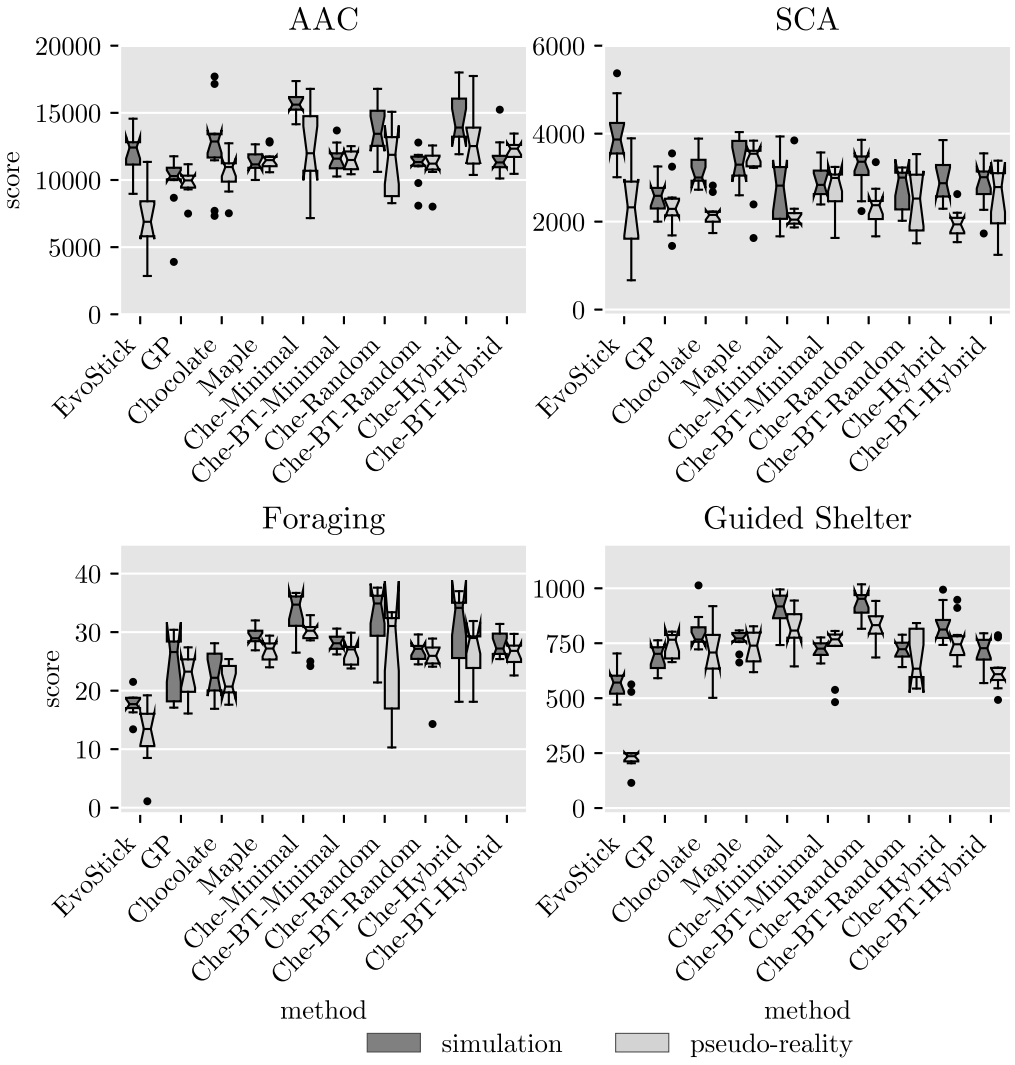


Figure adapted from Kuckling et al. (2020a).

Figure A.10: Results for all design methods for a budget of 25k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

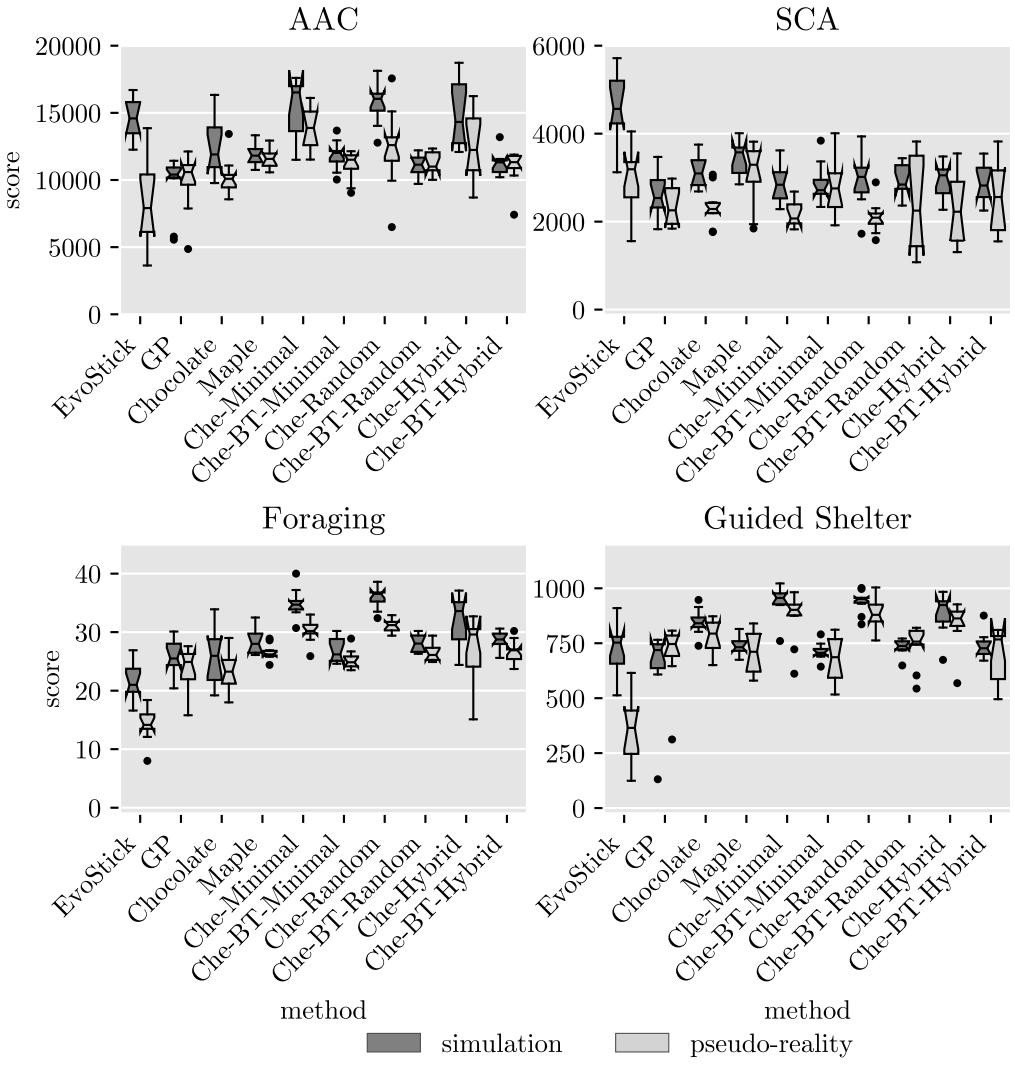


Figure adapted from Kuckling et al. (2020a).

Figure A.11: Results for all design methods for a budget of 50k. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.

Table A.1: The e-puck reference model RM2.2 used in `Cedrata` (Hasselmann et al., 2018).

Sensors	Variables
Proximity	$prox \in [1, 8]$, $\angle q \in [0, 2\pi]$
Ground	$gnd \in \{0, 0.5, 1\}$
Range-and-bearing	$n \in \mathbb{N}$, $r \in [0.5, 20]$, $\angle b \in [0, 2\pi]$
	n_s, r_s, $\angle b_s$, for $s \in \{1, ..., 6\}$

Actuators	Variables
Signal broadcast	$s \in \{0, 1, ..., 6\}$
Wheels	$v_l, v_r \in [-v, v]$, with $v = 0.16\,\mathrm{m/s}$
Control cycle period: $100\,\mathrm{ms}$	

I developed AutoMoDe-`Cedrata`, an automatic modular design method that contained behavioral modules that could return *success* and *failure*. `Cedrata` designs control software for a the e-puck robot, formalized through reference model RM2.2 (see Section A.4.1). On the basis of this reference model, I designed seven behavioral and seven conditional modules (see Section A.4.2). `Cedrata` then assembled these modules into behavior trees with a less restricted architecture (see Section A.4.3). `Cedrata` uses Iterated F-race (see Section 3.1) as the optimization algorithm. I also defined `Cedrata-GP` and `Cedrata-GE`, two variants of `Cedrata` that were using genetic programming (Koza, 1992) and grammatical evolution (O'Neill and Ryan, 2003) as optimization algorithms, respectively, but where otherwise identical to `Cedrata` (see Section A.4.5).

A.4.1 Reference model

The reference model RM2.2, on which `Cedrata` is based, is shown in Table A.1 (Hasselmann et al., 2018). RM2.2 also targets the e-puck robot (see Section 3.2) but provides a different form of access than RM1.1 to its sensors and actuators. The proximity sensors can detect obstacles up to $30\,\mathrm{cm}$ away, the ground sensors can sense the floor color on a grey scale and the range-and-bearing board can transmit messages up to $50\,\mathrm{cm}$. The control software can set the speed of the two wheels of the robot independently. It also always sends a signal value s, that can be equal to 0, which is a special value that means *no signal* and that is sent by default, or an integer in $\{1, ..., 6\}$. Similar to Hasselmann and Birattari (2020), signal values do not have a particular semantic, instead, it is the role of the design process to assign semantics to the signals. For the sensors, the reference model provides an aggregated vector (in the form of magnitude and direction) over all

Table A.2: Behavior and condition modules and their parameters used in `Cedrata`.

Behavior	Short	Parameters
Exploration	Exp	τ
Stop	Stop	
Grouping	Group	N_{max}, N_{min}, α
Isolation	Isol	N_{max}, N_{min}, α
Meeting	Meet	s, d_{min}
Acknowledgement	Ack	s, t_{max}
Emit Signal	ESig	s
Condition	**Short**	**Parameters**
Black Floor	Bflr	β
Grey Floor	Gflr	β
White Floor	Wflr	β
Neighborhood Count	Ngb	η, ξ
Inverted Neighborhood Count	INgb	η, ξ
Fixed probability	FP	β
Receiving signal	RSig	s

proximity readings and a single aggregated ground reading. The reference model also provides access to the number of neighboring robots n and a vector to their center of mass. Similarly, it provides the number of messaging robots and a vector to the center of mass of the messaging robots, for each signal $s \in S$. The control cycle period is $100\,\mathrm{ms}$, that is, every $100\,\mathrm{ms}$ the sensors are updated and the control software is invoked, generating a new tick in the behavior tree.

A.4.2 Modules

Based on the reference model RM2.2, I defined fourteen modules—seven behavior modules and seven condition modules. In the following descriptions of the signal-based conditions and behaviors, the set of signals $\{1, ..., 6\}$ will be denoted S. Some modules can use a special value any that is activated if any of the signals in S is received. The set $S^* = S \cup \{any\}$ will denote the sets used by these modules. The design process is free to choose several instances of the same module in an instance of control software and can tune the parameters independently for each instance of a module.

Behaviors are associated to action nodes and allow the robot to interact with the environment. The action nodes can return *success* or *failure* if the behavior

ends in a state that it considers being a success or a failure. Otherwise, they return *running*. The behavior modules are defined as follows:

Exploration The robot performs a random walk. It moves straight until it perceives an obstacle in front of itself. Then the robot turns on the spot for a random number of time steps in $\{0, ..., \tau\}$, where $\tau \in \{1, ..., 100\}$ is a tunable parameter. This behavior always returns *running*.

Stop The robot stays still. This behavior always returns *running*.

Grouping The robot tries to get closer to its neighbors by moving towards the geometric center of its neighbors. If the number of neighbors becomes greater than N_{max}, the behavior returns *success*, where N_{max} is a tunable parameter. If the number of neighbors becomes smaller than N_{min}, the behavior returns *failure*, where N_{min} is a tunable parameter. Otherwise, it returns *running*. The speed of convergence is controlled by the tunable parameter $\alpha \in [1, 5]$. The robot moves in the direction $w = w' - kw_0$, where w' is the target component and kw_0 is the obstacle avoidance component. If robots are perceived, then $w' = w_{r\&b} = (\alpha \cdot r, \angle b)$, otherwise $w' = (1, \angle 0)$. kw_0 is the obstacle avoidance component, with k being a constant fixed to 5 and w_0 defined as $w_0 = (prox, \angle q)$.

Isolation The robot tries to move away from its neighbors by moving in the opposite direction of the geometric center of its neighbors. If the number of neighbors becomes smaller than N_{min}, the behavior returns *success*, where N_{min} is a tunable parameter. If the number of neighbors becomes greater than N_{max}, the behavior returns *failure*, where N_{max} is a tunable parameter. Otherwise, it returns *running*. The speed of divergence is controlled by the tunable parameter $\alpha \in [1, 5]$. The Isolation behavior uses the same embedded collision avoidance as in Grouping, but with w' defined as: $w' = -w_{r\&b}$ if robots are perceived, where $w_{r\&b}$ is defined as in the Grouping behavior. Otherwise $w' = (1, \angle 0)$.

Meeting The robot listens for a signal $s \in S^*$ emitted by other robots and moves towards the geometrical center of the emitters. The behavior returns *success* if the distance between the robot and the geometrical center is smaller than a distance d_{min}, where d_{min} is a tunable parameter. The behavior returns *failure* if the robot does not perceive any robot sending the expected signal. Otherwise, the behavior returns *running*. The Meeting behavior uses the same embedded collision avoidance as in Grouping, but with w' defined as: $w' = w_{r\&b} = (\alpha \cdot r_s, \angle b_s)$ if robots emitting s are perceived. Otherwise $w' = (1, \angle 0)$.

Acknowledgement The robot sends a signal $s \in S$ and waits for an answer in the form of the same signal, where s is a tunable parameter. The behavior returns *success* if the signal is received or *running* if not. After t_{max} ticks, the behavior returns *failure* if the signal is still not received, where t_{max} is a tunable parameter. This behavior also sets the velocity of both wheels to zero.

Emit Signal The robot sets its emitted signal to $s \in S$ for the current tick, where s is a tunable parameter. This behavior always returns *success*. This behavior also sets the wheel velocity to zero.

Conditions are associated to condition nodes and check an aspect of the environment. The condition nodes return *success*, when their condition is met, or *failure*, otherwise. The condition modules are defined as follows:

Black Floor When all ground sensors detect a black floor, the condition returns *success* with probability β, where β is a tunable parameter.

Grey Floor When all ground sensors detect a grey floor, the condition returns *success* with probability β, where β is a tunable parameter.

White Floor When all ground sensors detect a white floor, the transition is enabled with probability β, where β is a tunable parameter.

Neighborhood Count Returns *success* with probability $z(n) = \frac{1}{1+e^{\eta(\xi-n)}}$, where n is the number of robots in the neighborhood, $\xi \in \{0, 1, ..., 10\}$ and $\eta \in [0, 20]$ are tunable parameters.

Inverted Neighborhood Count Same as Neighborhood Count but with probability $1 - z(n)$.

Fixed Probability Returns *success* with probability β, where β is a tunable parameter.

Receiving Signal Returns *success* if the robot has perceived a neighbor sending $s \in S^*$ in the last 10 ticks, where s is a tunable parameter.

A.4.3 Control architecture

In `Cedrata`, the optimization process can create a tree that has a maximum of three levels and a maximum of three children per node. The top-level node must be a control-flow node. Nodes of the second level can be control-flow nodes, action nodes or condition nodes. If it is an action node or a condition node, then the node can have no children itself. Not all branches are forced to have the same depth:

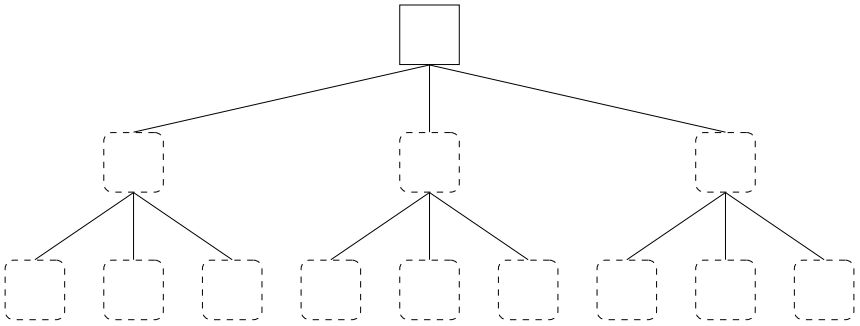


Figure originally published in Kuckling et al. (2022).

Figure A.12: The possible behavior tree structure for `Cedrata`. In `Cedrata`, the top-level node can be any control-flow node. Underneath it the tree can have between one and three nodes, chosen among control-flow nodes, action nodes and condition nodes. If a control-flow node is chosen, then it can have between one and three children, which are either action nodes or condition nodes.

the top-level node could have some children that are control-flow nodes and some that are action or condition nodes. Nodes on the third level can only be action nodes or condition nodes. The structure of such trees is depicted in Figure A.12. The optimization process can choose any control-flow node type to be either a sequence, sequence*, selector or selector* node. For a formal definition of these nodes, see Marzinotto et al. (2014). Prior research has shown that high complexity in automatic design methods can increase the difficulties in crossing the reality gap (Francesca et al., 2014, 2015; Hasselmann et al., 2021). In order to match the complexity of other AutoMoDe methods (Kuckling et al., 2018a), the tree may have at most four action nodes and four condition nodes. The constraints on the depth and on the number of children implicitly impose that the tree contains no more than four control nodes.

A.4.4 `Cedrata-GP` and `Cedrata-GE`

The optimization algorithm of `Cedrata` is Iterated F-race (see Section 3.1). However, other researchers have used genetic programming or grammatical evolution to design behavior trees (Jones et al., 2018b; Neupane and Goodrich, 2019). Therefore, I also developed `Cedrata-GP` and `Cedrata-GE` that are using genetic programming and grammatical evolution, respectively, as their optimization algorithm. `Cedrata-GP` and `Cedrata-GE` use the same reference model, modules and architecture as `Cedrata`. They differ only in the optimization algorithm employed.

`Cedrata-GP` uses genetic programming (Koza, 1992) as the optimization

Table A.3: Parameters for genetic programming and grammatical evolution. Parameters for genetic programming are those used in the work of Jones et al. (2018b) and parameters for grammatical evolution are those used in the work of Neupane and Goodrich (2019).

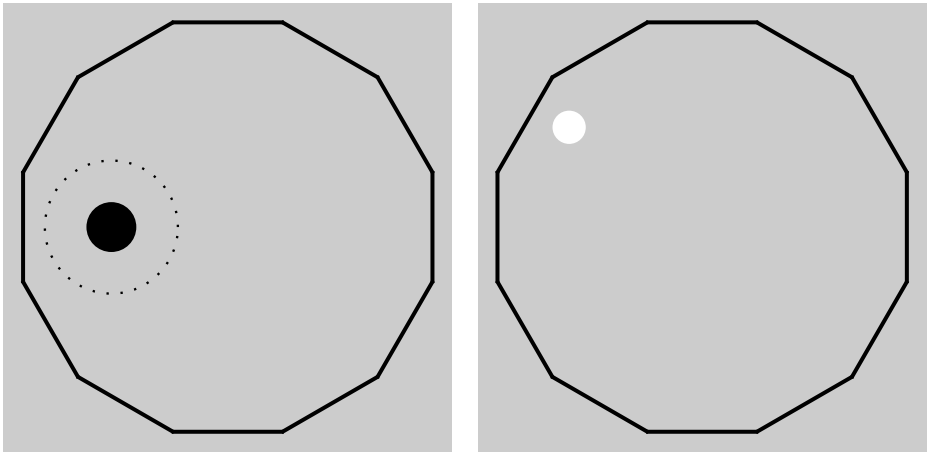
Parameter	Genetic Programming	Grammatical Evolution
Initialization	half-and-half	uniform-tree
Selection strategy	tournament selection	truncation
Tournament size	3	–
Selection proportion	–	50%
Crossover	one-point crossover	one-point crossover
Population size	25	100
Number of elites	3	1
Crossover probability	0.8	0.9
Uniform mutation probability	0.05	–
Shrink mutation probability	0.1	–
Node replacement mutation probability	0.5	–
Ephemeral mutation probability	0.5	–
Flip per codon mutation probability	–	0.01
Codon size	–	1000

algorithm. The parameters of this design method are those used in the work of Jones et al. (2018b) and summarized in Table A.3. We use the genetic programming implementation of the DEAP library (Fortin et al., 2012).

`Cedrata-GE` uses grammatical evolution (O'Neill and Ryan, 2003) as the optimization algorithm. The parameters of this design method are those used in the work of Neupane and Goodrich (2019) and summarized in Table A.3. We use the grammatical evolution implementation of PonyGE2 (Fenton et al., 2017).

A.4.5 Experiments

I validated `Cedrata`, `Cedrata-GP`, and `Cedrata-GE` on a set of two missions: MARKER AGGREGATION and STOP. Both missions were performed in a dodecagonal arena (see Figure A.13) and lasted $250\,\mathrm{s}$. All code and data is available from the supplementary material (Kuckling, 2023).



(a) MARKER AGGREGATION (b) STOP

Figure originally published in Kuckling et al. (2022).

Figure A.13: Layouts of the arena for the missions considered.

MARKER AGGREGATION

In the mission MARKER AGGREGATION (see Figure A.13a), the robots had to aggregate within the dotted area. The area itself was not perceivable by the robots. Instead, a black spot was placed in the middle of the aggregation area that can serve as a marker. The objective function for this mission was the cumulative time that the robots spend within the aggregation area:

$$F_{MA} = \sum_{i=0}^{2500} N_A^i, \tag{A.3}$$

where N_A^i is the number of robots in the aggregation area at time step i. The higher the score of the objective function, the better the robots performed the mission.

STOP

In the mission STOP (see Figure A.13b), the robots had to find a white spot and then stop as soon as possible. A robot was considered moving, if it had traveled more than $5\,\mathrm{mm}$ in the last time step. The objective function for this mission was reduced for each robot that was not moving at any given time step before the white spot has been found and for each robot that was moving after the white spot had been found and additionally for the time that the swarm needed to discover the

white spot:

$$F_{Stop} = 100000 - \left(\bar{t}N + \sum_{t=1}^{\bar{t}} \sum_{i=1}^{N} \bar{I}_i(t) + \sum_{\bar{t}}^{2500} \sum_{i=1}^{N} I_i(t) \right), \tag{A.4}$$

where $\bar{t}$ is the time step during which the white spot was discovered, $I_i(t)$ is an indicator that a robot i has moved in time step t and $\bar{I}_i(t)$ is an indicator that a robot i has not moved in time step t. The higher the score of the objective function, the better the robots perform the mission.

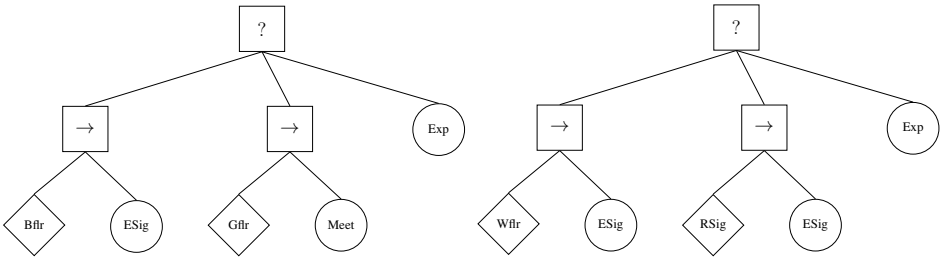
Design methods

I considered `Cedrata`, `Cedrata-GP`, and `Cedrata-GE`, as described in this chapter. I also included several manual designs. For the manual designs, I asked human designers—with prior experience in swarm robotics, but not with behavior trees—to design control software within the same constraints as `Cedrata`, that is, with the same modules and architecture. The human designers had access to the AutoMoDe Editor (Kuckling et al., 2021a), a tool that allows the designers to visualize and manipulate the behavior trees and to launch simulations of the designed behavior tree. The human designers received feedback about their designed behavior tree through the objective function and a visual representation of the arena and the behavior of the swarm.

Lastly, I included a reference design as an additional point of reference for the reader. These reference designs were not part of the experimental protocol. They were not optimized and did not aim to be the best performing solutions for each mission, but simply to provide a sensible solution. These designs served to highlight particular strategies that I expected to be discovered in each mission. They were not known to the human designers prior to their manual designs.

Reference designs

The reference design for the mission MARKER AGGREGATION is shown in Figure A.14a. In this design, robots explored the arena until they found the marker. Then, using the signal framework, they attracted their neighbors to the aggregation area. At any given time step, the tick traversed the three subtrees from left to right. The left subtree handled the case where the robot is on the marker. If the condition Black Floor evaluated true, then the tick was passed on to the action node, which invoked the Emit Signal behavior. Since Emit Signal always returns *success* and the action node is the last child of the sequence node, this subtree then returned *success* as well. This caused the selector node to also return *success*. If the condition Black Floor is not met, then the tick is passed into the middle subtree,



(a) MARKER AGGREGATION (b) STOP

Figure originally published in Kuckling et al. (2022).

Figure A.14: The reference designs for the two missions. The conditions and actions names have been abbreviated in the following way: Exp: Exploration; Meet: Meeting; ESig: Emit Signal; Bflr: Black Floor; Gflr: Grey Floor; Wflr: White Floor; RSig: Receiving Signal.

which handled the case where the robot was on the grey floor and perceived at least one signaling neighbor. Here, if the condition Grey Floor is met, the robot executed one time step of the Meeting behavior. If Meeting returned *success* or *running*, then the tick left the tree. If either Meeting or Grey Floor returned *failure*, then the tick was passed to the last subtree. This subtree only consists of an action node with the Exploration behavior.

The reference design for the mission STOP is shown in Figure A.14b. In this design, robots sent and forwarded signals to their neighbors to transmit the information that the white spot had been discovered. If a robot received a signal, it stopped; if it did not receive any signal, it explored the arena to find the white spot. At any given time step, the tick traversed the three subtrees from left to right. The left subtree handled the case in which the robot was on the white spot. While the condition White Floor evaluated true, the robot executed the behavior Emit Signal to signal the other robots the discovery of the spot. If the condition White Floor was not met, then the tick was passed to the middle subtree that forwarded received signals. If the condition Receiving Signal was met, then the tick was passed to the Emit Signal behavior that emitted the same signal as is checked for in the Receiving Signal condition. If the Receiving Signal condition was not met, then the tick was passed to the right subtree, which consisted only of an action node with the Exploration behavior.

Protocol

For each mission, `Cedrata` was executed with different budgets: $20\,000$, $50\,000$, $100\,000$ and $200\,000$ simulation runs. The budget specified the number of simulations that the design process was allowed to perform before it returned the best

control software produced. Additionally, Cedrata-GP and Cedrata-GE were tested on a budget of 200 000 simulation runs. For each combination of method, mission and budget, 10 independent runs of the methods were performed, leading to 10 instances of control software. The manual designs were done by four human designers per mission, with a maximum design duration of 4 hours.

Simulations were performed in a realistic and physics-based simulation environment (see Section 3.6). The generated instances of control software of all designs methods are assessed in pseudo-reality to investigate the impact of the reality gap.

A.4.6 Results

Figure A.15 shows the results for the missions STOP and MARKER AGGREGATION. Results are shown for both the performance in simulation and pseudo-reality.

In the mission MARKER AGGREGATION, there is a clear trend of increasing performance for Cedrata with increasing budget. A detailed investigation of the generated control software revealed that Cedrata developed two general solution strategies: one strategy was based on the communication framework, while the other was not. In the communication-less strategy (for an example, see Figure A.16a), the robots explored the arena until they discovered the black spot, at which point they usually stopped. In the communication-based strategy (see Figure A.16b), however, the robots made use of the communication behaviors to quickly aggregate within the target area. The communication-based designs were similar in that regard to the reference design. The performance of Cedrata for each budget then seemed to primarily depend on the ratio of the two strategies. Indeed, for design budgets of 20 000 and 50 000 simulation runs, Cedrata only produced control software that uses the communication-less strategy. For a budget of 100 000 simulation runs, Cedrata produced a single solution that followed the communication-based strategy and for a budget of 200 000 simulation runs, four designs made use of that strategy. It appears that the ratio of communication-less to communication-based strategies depended on the available budget. Indeed, as the communication-based strategy required at least two modules to interact correctly, Iterated F-race is more likely to discover such a combination the more often it sampled new solutions, which depended on the number of iterations and therefore the budget.

When comparing all considered design methods, the manual designers all found solutions that made use of communication. Their control software performed similar well as the communication-based behavior trees generated by Cedrata and better than the reference design, which was not meant to be the best performing solution, but just to highlight the general strategy. The human designers were therefore not only able to discover the strategy but also to find a reasonable tuning

for the parameter.

Cedrata-GP and Cedrata-GE both failed to generate any solution making use of the communication modules, even for a budget of $200\,000$ simulation runs. Interestingly, both design methods generated solutions that, under the right circumstances, performed nearly as good as the best instances of control software generated by Cedrata. However, this appears to be mostly due to the initial starting position favoring quick aggregation within the target zone and in total both Cedrata-GP and Cedrata-GE performed worse than Cedrata.

Unlike in the mission MARKER AGGREGATION, there was no improvement for increasing budgets in the mission STOP. Instead, the performance remained relatively stable. Investigation of the generated behavior trees revealed that Cedrata failed to make use of the communication modules for this mission. All generated behavior trees employed a strategy, where the robots were using the Isolation behavior (for an example, see Figure A.16c). As a result, the swarm expanded and, with high probability, a robot passed over the white spot. At the end of the expansion phase, the robots slowed down and moved relatively little, often falling below the threshold of $5\,\mathrm{mm}$ per time step. Some behavior trees also included an Exploration module for cases when no neighbors were detected.

Comparing all design methods showed that the manual designs, just like the reference design, made use of the communication framework and showed the best performance. Both Cedrata-GP and Cedrata-GE found solutions that followed the same Isolation-based strategy as Cedrata and achieved similar performances. For all design methods, there were some runs where the performance was relatively close to 0. Often, in these runs, the control software failed to find the white spot.

I made some observations that held for all considered missions: The first observation is that all design methods showed a relatively small pseudo-reality gap. That is, they experience only a small drop in performance when assessing the control software in pseudo-reality. I believe that this is a first indicator that Cedrata and the design methods based on it might transfer well into reality as well. A second observation is that all behavior trees generated by Cedrata, Cedrata-GP and Cedrata-GE contained many modules that never were ticked by the behavior tree. I believe this to be because of the reduced restrictions in the architecture, which allowed modules to be easily placed in the tree in a way that ensured they would never receive a tick. The design process had no explicit way of distinguishing necessary and superfluous modules and all techniques that aim at generating new behavior trees (random sampling around elites, cross-overs, mutations) were therefore highly likely to transfer some of the superfluous modules into the newly generated behavior tree. This poses a challenge to the automatic design process. Namely, that the design process will spend some resources on tuning these superfluous modules, which have no influence on the behavior of the swarm, thus

effectively wasting a part of the allocated budget. Lastly, I observed that the automatic design process had difficulties generating communication-based behaviors. In both missions, MARKER AGGREGATION and STOP, the human designers found well performing solutions that made use of the communication framework. Only in the mission MARKER AGGREGATION was `Cedrata` able to generate at least a few solutions following a similar strategy. My initial hypothesis was that this might have been caused by some properties of the underlying optimization algorithm, Iterated F-race. I have therefore replaced Iterated F-race with two different optimization algorithms, whose parametrization we have taken from other works in the swarm robotics literature. Unfortunately, both `Cedrata-GP` and `Cedrata-GE` appeared to have even greater difficulties generating communication-based behaviors than `Cedrata`. I believe that this could be due to the fact that communication requires two corresponding modules, a sender and a receiver, while all other strategies can rely on a single module.

A.4.7 Discussion

The results generated by the human designers showed that the modules and constraints of `Cedrata` were sensible, as the human designers were able to design control software that performed satisfactorily. Furthermore, as the human designers had no prior experience with behavior trees, this seems to be an indicator that behavior trees are an intuitive control architecture to design for. The automatic design method `Cedrata`, on the other hand, was not able to generate communication-based behaviors. I hypothesized that this might have been due to some property of the optimization algorithm Iterated F-race, and therefore I created `Cedrata-GP` and `Cedrata-GE`, two variants of `Cedrata` that were based on genetic programming and grammatical evolution, respectively. Neither of these two variants was able to generate communication-based strategies either.

For future work, I would like to investigate in more detail how an automatic design process can discover meaningful communication-based strategies and why the approach taken in this work failed. The results of this work indicated that simply tuning the parameters of an optimization algorithm would probably not be enough. Nevertheless it would be interesting to investigate the effects of different parameters on the performance of generated solutions, especially with respect to the exploration-exploitation trade-off. Another issue for investigation could be the mapping of behavior trees into representations that can be manipulated by the genetic programming and grammatical evolution implementations. One possible approach to create communication-based behaviors could be to create an interleaved optimization process. Starting from a minimal communicating solution, the design process alternates between fixing the sending or the receiving part of the behavior tree and optimizing the remaining part of the tree. Another approach to solve

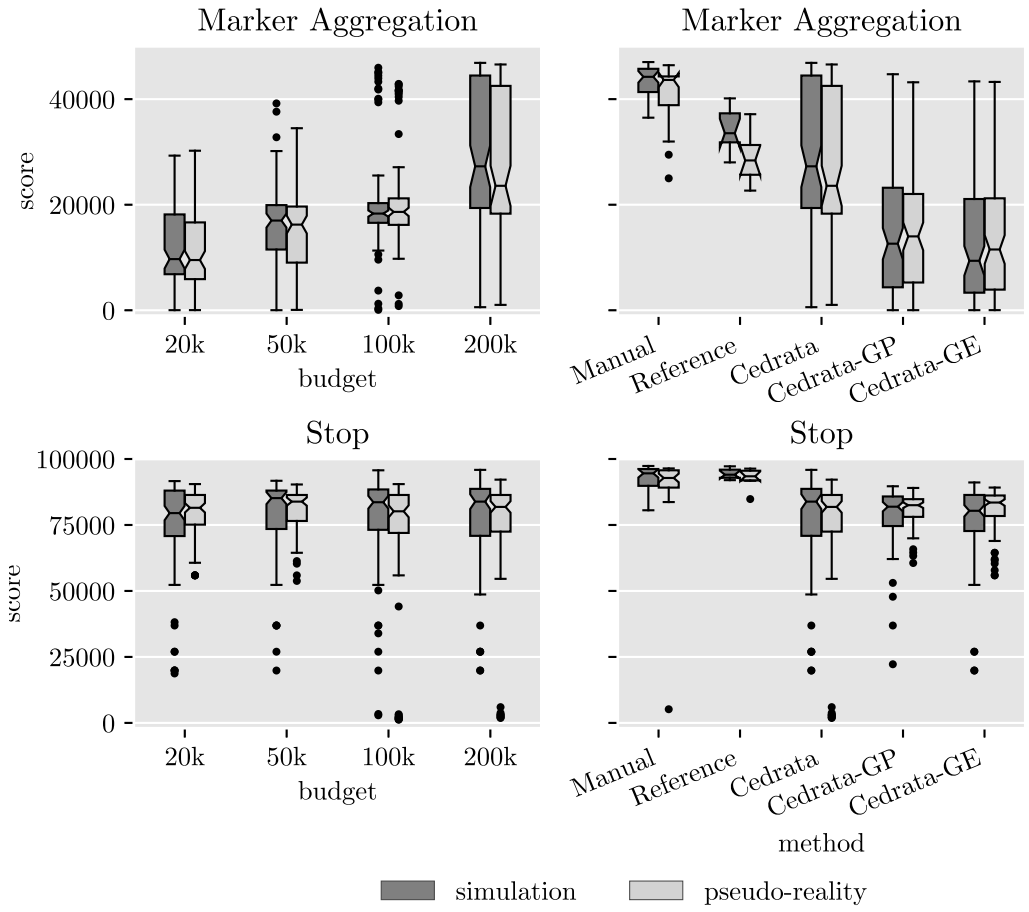
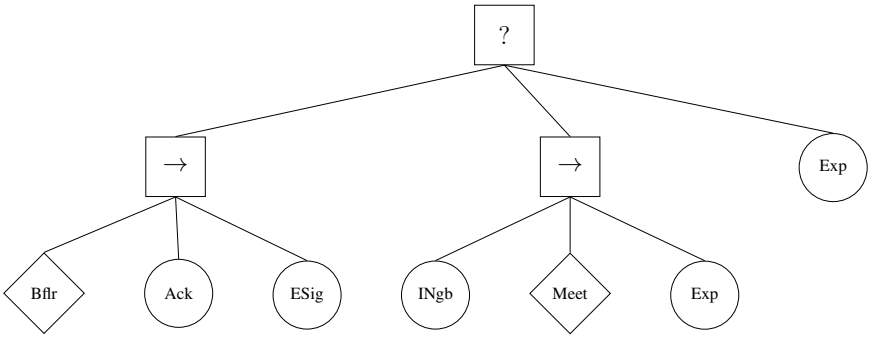
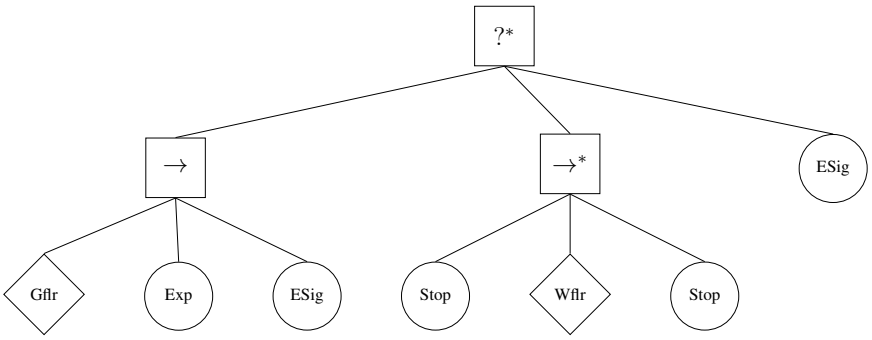


Figure adapted from Kuckling et al. (2022).

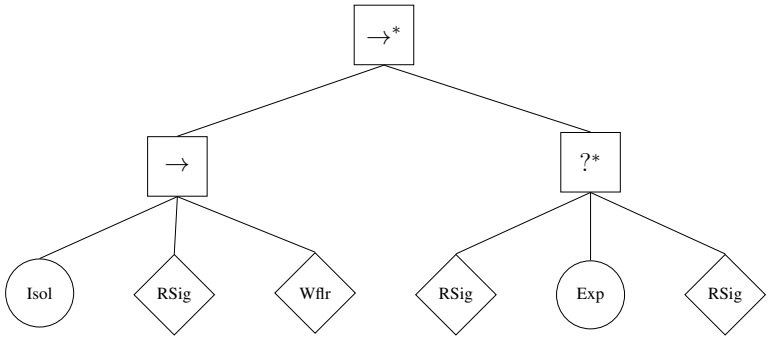
Figure A.15: Results for the mission MARKER AGGREGATION (top) and STOP (bottom). The left plots show the development of the performance over increasing budget for `Cedrata`. The right plots show the comparison of all design methods under consideration for a budget of 200 000 simulation runs. The dark grey boxes represent performance obtained in the design context, light grey boxes represent performance obtained in the pseudo-reality context.



(a) MARKER AGGREGATION, without communication



(b) MARKER AGGREGATION, communication-based



(c) STOP

Figure originally published in Kuckling et al. (2022).

Figure A.16: Typical behavior trees generated by `Cedrata`.

this problem could be cooperative co-evolution. One could possibly create two distinct populations that are given a sending or receiving module, respectively. This ensures the existence of communication from the starting population. Subsequent generations could then refine the communication protocol and integrate it with the other modules. Additionally, the results presented here showed that `Cedrata` and its variants were able to perform satisfactorily also in pseudo-reality. While this is an indicator that the design approaches might cross the reality gap well, I would like to confirm this hypothesis by performing real robot experiments.

Appendix B

Search space considerations

B.1 Search space size for AutoMoDe-`Chocolate` and AutoMoDe-`Maple`

B.1.1 Search space for finite-state machines

General case

Consider a finite-state machine with up to s_{max} states. Each state has at least one and at most t_{max} outgoing transitions that cannot point back into the same state they origin from. The number of all such finite-state machines (written $|S_{FSM}|$) can be described by equation B.1:

$$\begin{aligned} |S_{FSM}| &= \sum_{s=1}^{s_{max}} |S_{FSM}(s)| \\ &= |S_{FSM}(1)| + \sum_{s=2}^{s_{max}} |S_{FSM}(s)|. \end{aligned} \tag{B.1}$$

$|S_{FSM}(s)|$ describes the number of possible finite-state machines with exactly s states. The trivial case of $s = 1$ needs to be handled separately as a finite-state machine with exactly one state does not have any transitions. There are exactly $|S_{FSM}(1)| = |\mathcal{B}|$ finite-state machines with exactly one state, as only the behavior for the single state can be chosen. For more than one state $s > 1$, the number of possible finite-state machines can be described by the number of s (independent) choices, one for each state. This can be modeled by choosing s times independently from the set of all possible states (*multiplication principle*). In this model, the state already contains information about all outgoing transitions (e.g. their number, the

target, and the condition).

$$|S_{FSM}(s)| = |S_{state_1}(s)| * |S_{state_2}(s)| * ... * |S_{state_s}(s)| = \prod_{i=1}^{s} |S_{state}(s)|. \quad \text{(B.2)}$$

If the number of states in a finite-state machine is fixed to s, then number of configurations for each state is the same, and can be expressed by $|S_{state}(s)|$. This is because each state is composed of the behavior (chosen from $\mathcal{B}$) and up to t_{max} outgoing transitions to the $s-1$ other states.

$$|S_{state}(s)| = \sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * \prod_{j=1}^{t} |S_{transition}(s)| \right). \quad \text{(B.3)}$$

The number of possible outgoing transition is defined by the (independent) choice of a condition (from the set $\mathcal{C}$) and the target state. The target state can be modeled by a mapping from the states unto the set $\{1, 2, ..., s-1\}$. Therefore there are $s-1$ possible target states for the transitions.

$$|S_{transition}(s)| = |\mathcal{C}| * (s-1). \quad \text{(B.4)}$$

By substituting the equation for the number of outgoing transitions of a state (equation B.4) into the equation for the number of possible configurations of a state (equation B.3) we obtain:

$$\begin{aligned} |S_{state}(s)| &= \sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * \prod_{j=1}^{t} (|\mathcal{C}| * (s-1)) \right) \\ &= \sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * |\mathcal{C}|^t * (s-1)^t \right). \end{aligned} \quad \text{(B.5)}$$

Substituting the result obtained in equation B.5 into equation B.2 then leads to:

$$\begin{aligned} |S_{FSM}(s)| &= \prod_{i=1}^{s} \sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * |\mathcal{C}|^t * (s-1)^t \right) \\ &= \left(\sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * |\mathcal{C}|^t * (s-1)^t \right) \right)^s. \end{aligned} \quad \text{(B.6)}$$

Finally after substituting equation B.6 into equation B.1, we obtain an equation for the size of the search space for all finite-state machines with up to s_{max} states and t_{max} outgoing transitions per state:

$$|S_{FSM}| = |\mathcal{B}| + \sum_{s=2}^{s_{max}} \left(\sum_{t=1}^{t_{max}} \left(|\mathcal{B}| * |\mathcal{C}|^t * (s-1)^t \right) \right)^s. \quad \text{(B.7)}$$

`Chocolate`

In `Chocolate` the finite-state machines are limited to four states and each state can only have up to four outgoing transitions. Inserting the values $s_{max} = 4$ and $t_{max} = 4$ into equation B.7 leads to the following equation:

$$\begin{aligned}
\#S(FSM) = {} & 43046721\,|\mathcal{B}|^4\,|\mathcal{C}|^{16} + 57395628\,|\mathcal{B}|^4\,|\mathcal{C}|^{15} + 47829690\,|\mathcal{B}|^4\,|\mathcal{C}|^{14} \\
& + 31886460\,|\mathcal{B}|^4\,|\mathcal{C}|^{13} + 16474671\,|\mathcal{B}|^4\,|\mathcal{C}|^{12} + 7085880\,|\mathcal{B}|^4\,|\mathcal{C}|^{11} \\
& + 2598156\,|\mathcal{B}|^4\,|\mathcal{C}|^{10} + 787320\,|\mathcal{B}|^4\,|\mathcal{C}|^{9} + 203391\,|\mathcal{B}|^4\,|\mathcal{C}|^{8} \\
& + 43740\,|\mathcal{B}|^4\,|\mathcal{C}|^{7} + 7290\,|\mathcal{B}|^4\,|\mathcal{C}|^{6} + 972\,|\mathcal{B}|^4\,|\mathcal{C}|^{5} + 81\,|\mathcal{B}|^4\,|\mathcal{C}|^{4} \\
& + 4096\,|\mathcal{B}|^3\,|\mathcal{C}|^{12} + 6144\,|\mathcal{B}|^3\,|\mathcal{C}|^{11} + 6144\,|\mathcal{B}|^3\,|\mathcal{C}|^{10} + 5120\,|\mathcal{B}|^3\,|\mathcal{C}|^{9} \\
& + 3072\,|\mathcal{B}|^3\,|\mathcal{C}|^{8} + 1536\,|\mathcal{B}|^3\,|\mathcal{C}|^{7} \\
& + 640\,|\mathcal{B}|^3\,|\mathcal{C}|^{6} + 192\,|\mathcal{B}|^3\,|\mathcal{C}|^{5} + 48\,|\mathcal{B}|^3\,|\mathcal{C}|^{4} + 8\,|\mathcal{B}|^3\,|\mathcal{C}|^{3} + |\mathcal{B}|^2\,|\mathcal{C}|^{8} \\
& + 2\,|\mathcal{B}|^2\,|\mathcal{C}|^{7} + 3\,|\mathcal{B}|^2\,|\mathcal{C}|^{6} + 4\,|\mathcal{B}|^2\,|\mathcal{C}|^{5} + 3\,|\mathcal{B}|^2\,|\mathcal{C}|^{4} + 2\,|\mathcal{B}|^2\,|\mathcal{C}|^{3} \\
& + |\mathcal{B}|^2\,|\mathcal{C}|^{2} + |\mathcal{B}|
\end{aligned} \tag{B.8}$$

In this equation, each summand $x\,|\mathcal{B}|^b\,|\mathcal{C}|^c$ indicates the number of possible finite-state machines with b states and a total of c transitions. The term x denotes the number of possible finite state machines that combine c transitions and b states with the restriction of at most t_{max} outgoing transitions per state.

For example $2\,|\mathcal{B}|^2\,|\mathcal{C}|^7$ denotes that `Chocolate` can generate 2 different topologies of finite-state machines containing 2 states and 7 outgoing transitions. Because of the limit of $t_{max} = 4$ outgoing transitions per state, 3 of the 7 outgoing transitions need to be associated to one state, and the 4 remaining transitions to the other state. However each of the two states can have the four transitions, leading to two possible distributions.

Summing the coefficients of equation B.8 results in a total of $207\,387\,017$ different topologies of finite-state machines.

The defining factor for the size of the search space is however the search space defined by the modules. Even without taking the parameters into account, there are $6^{16} = 2\,821\,109\,907\,456$ possible ways of assigning a condition to each transition in the case of the maximum number of states and transitions. We can therefore conclude that the size of the search space for `Chocolate` is in $O\left(|\mathcal{B}|^4\,|\mathcal{C}|^{16}\right)$.

B.1.2 Search space for behavior trees

General case

Consider a behavior tree with depth d, that is, $d+1$ nodes on the longest path from the (implicitly) defined root node to a leaf node. Additionally, let the top-level node (only child of the root), be at level 1. All of its children are at level 2, their children at level 3, and so on. In this case the level of a node is equivalent to its depth in the tree.

Suppose that we fix a level i. On this level i we can choose either a control-flow node out of a subset of all possible control-flow nodes $\mathcal{N}_i \in \mathcal{N}$, or a leaf node (either action or condition node). Additionally, every control-flow node on level i must have between c_{min} and c_{max} children.

Let $BT_{=l}$ be the set of behavior trees with a depth of exactly l. That is the there are exactly l nodes from the top-level node to the furthest leaf node.

Similarly let $BT_{<l}$ be the set of behavior trees where there exists no path between the top-level node and any leaf node that has at least l nodes in it. It should be noted that the following equality holds true:

$$BT_{<l} = \bigcup_{i=1}^{l-1} BT_{=i}. \tag{B.9}$$

The last important notation is $BT_{\leq l}$, the set of all behavior trees with a depth of at most l. The following two equalities hold up:

$$BT_{\leq l} = BT_{<l+1} \tag{B.10}$$

$$BT_{\leq l} = BT_{=l} \cup BT_{<l}. \tag{B.11}$$

The number of behavior trees with at most l levels can be described by the following recursive formula:

$$|BT_{=1}| = |\mathcal{B}| + |\mathcal{C}| \tag{B.12}$$

$$\begin{aligned} |BT_{\leq l+1}| &= |BT_{=l+1}| + |BT_{<l+1}| \\ &= |BT_{=l+1}| + |BT_{\leq l}| \\ &= \sum_{i=l+1}^{1} |BT_{=i}|\,. \end{aligned} \tag{B.13}$$

It should be noted that this formula covers the recursive anchor for $l = 1$ (the leaf nodes). If the restrictions applied to a behavior tree allow it, this recursive formula can also have a recursive anchor for $BT_{=i}, i > 1$.

For $|BT_{=i}|, i > 1$ we can show the following:

$$|BT_{=i}| = |\mathcal{N}_i| \sum_{c=c_{min}}^{c_{max}} \left(|BT_{=i-1}| + |BT_{<i-1}|\right)^c - |BT_{<i-1}|^c . \tag{B.14}$$

That is because no behavior trees with $i > 1$ levels can be a leaf node. Additionally they if they have a depth of i, they need to have at least one subtree under the top-level node with exactly $i - 1$ levels. It is however not necessary to only have a single subtree with these many levels. Indeed any number of subtrees (with at least more than one) are acceptable. By the inclusion-exclusion principle, we can include all behavior trees as subtrees that have either $l - 1$ or less then $l - 1$ levels ($(|BT_{=i-1}| + |BT_{<i-1}|)^c$) but we need to include the case that all subtrees have less then $l - 1$ levels ($|BT_{<i-1}|^c$). This needs to be done for all mutually exclusive choices for the number of children and all independent choices of the control-flow node.

Maple

In `Maple` we have a restricted version of the behavior trees, that can have exactly three levels. Because of the special restrictions, we can define a recursive anchor for $BT_{=2}$, describing our selector subtrees. There are $|\mathcal{C}| * |\mathcal{B}|$ possible combinations for the selector subtrees, because of the independent choices of the selector node (no true single choice), condition for the condition node and behavior for the action node.

$$\begin{aligned} |BT_{=2}| &= |\{?\}| * |\mathcal{C}| * |\mathcal{B}| \\ &= |\mathcal{C}| * |\mathcal{B}| . \end{aligned} \tag{B.15}$$

Additionally all other levels (in this case only the top-level) can have between $c_{min} = 1$ and $c_{max} = 4$ children. If we use these restrictions in equation B.14 it results in:

$$\begin{aligned} |BT_{=3}| &= |\mathcal{N}_3| \sum_{c=c_{min}}^{c_{max}} \left(|BT_{=2}| + |BT_{<2}|\right)^c - |BT_{<2}|^c \\ &= |\{\rightarrow^*\}| \sum_{c=1}^{4} \left(|BT_{=2}| + 0\right)^c - 0^c \\ &= \sum_{c=1}^{4} \left(|BT_{=2}|\right)^c \\ &= \sum_{c=1}^{4} \left(|\mathcal{C}| * |\mathcal{B}|\right)^c \\ &= |\mathcal{C}|^1 |\mathcal{B}|^1 + |\mathcal{C}|^2 |\mathcal{B}|^2 + |\mathcal{C}|^3 |\mathcal{B}|^3 + |\mathcal{C}|^4 |\mathcal{B}|^4 . \end{aligned} \tag{B.16}$$

Here again the coefficients of $x\,|\mathcal{C}|^i\,|\mathcal{B}|^i$ describe the number of ways it is possible to construct a restricted behavior tree with i. However there is just a single way of combining the subtrees (all under the top-level node).

B.2 Proofs of completeness for perturbation operators

Finite-state machines

Completeness of the perturbation operators

In this section, I provide proofs that any valid finite-state machine can be transformed into any other valid finite-state machine through the application of the perturbation operators P1 - P11 (see Section 4.3).

Corollary FSM.1 Given a valid finite-state machine, according to the criteria of this section, if a state has the maximum number of outgoing transitions, then at least two of its outgoing transitions point to the same end state.

Proof. In this setting, a valid finite-state machine has at most four states (see the definitions of validity further up). If a state has the maximum number of four outgoing transitions and no transition is allowed to be self-referencing, then each outgoing transition can point to one of the other states, which are at most three possibilities. The pigeonhole principle now states that there needs to be at least two of the four outgoing transitions pointing towards the same end state. □

Completeness of perturbation operators Let FSM and FSM' be two finite-state machines that are valid instances of control software according to the previous definition. FSM can be transformed into FSM' through the use of the perturbation operators defined in this section.

Proof. The proof of the aforementioned statement will be divided in several steps. After all steps have been executed, the initial finite-state machine FSM has been transformed into FSM'. The transformation steps are:
1. add states, if necessary;
2. transform into clique;
3. remove unneeded states, if necessary;
4. remove unneeded transitions, if necessary;
5. add transitions;
6. move initial state;

7. match modules;
8. match parameters.

Step 1 If FSM has fewer states than FSM': repeatedly apply the operator P3 (add state) to add states to FSM until it has the same number of states as FSM'. This also adds one incoming and one outgoing transition to each state. If a state needs to be added, but all other states already have the maximum number of outgoing transitions, select one state s and one transition t outgoing of s into s' in such a way that s has another transition to s' (this is possible because of Corollary FSM.1). Remove this transition t with the operator P2 (remove transition). Now add the additional state s'' using the operator P3 (add state), creating a transition from s to s' and a transition from s to any other state in the finite-state machine.

Step 2 For each ordered pair of states s, s' in the finite-state machine, add a transition from s to s' through the application of the operator P1 (add transition), if it does not already exist. If s already has the maximum number of outgoing transitions, then P1 (add transition) is not applicable. Instead, select one transition from s to s'', where s'' is another state such that at least two transitions point from s to s'' (possible because of Corollary FSM.1) and change its endpoint to s' through the application of P6 (move transition end). This step transforms the state transition graph of FSM into a directed clique.

Step 3 For every state, define a matching from the states of FSM' to the states of FSM. As Step 1 guarantees that FSM has at least the same number of states as FSM' and step 2 did not alter the number of states in FSM, every state of FSM' can be matched to a state in FSM. Excess states in FSM are not matched and are marked for deletion. Using operator P4 (remove state), remove every such state s that has been marked for deletion. This is possible, as the state is definitely not an articulation vertex, as the remaining states and their transitions still form a directed clique.

Step 4 For every state s in FSM and every outgoing transition of s, if FSM' has at least one transition from $s = start(t)$ to $end(t)$, then match t to one of these transitions. Else remove this transition through the application of the perturbation operator P2 (remove transition). This is possible, as this can never remove the last outgoing transition of s. Indeed the corresponding state in FSM' has at least one outgoing transition, or it is the only state in both finite-state machines FSM and FSM' so neither finite-state machine has a transition to remove anyway. Also, it cannot remove the only incoming transition into another state, as this transformation preserves at least one of the incoming transitions from s to s'. If

this matching would delete all incoming transitions into s', this would mean that there is no state s'' in FSM' either that has a transition to s', thus creating an invalid configuration.

Step 5 For every state s in FSM, if s has fewer outgoing transitions than its corresponding state in FSM', apply operator P1 (add transition) repeatedly, adding outgoing transitions to s until it has the same number of outgoing transitions as its corresponding state. Match this newly generated transition to a transition in FSM' that has no matching yet. Use perturbation operator P6 (move transition end) to move the end state of the newly created transition to match the end state of the corresponding transition in FSM'. This again is possible because the newly created transition could not have been the only incoming transition into its original end state, otherwise this state would have been in an invalid configuration after step 4.

Step 6 Use the perturbation operator P7 (change initial state) to move the initial state to the state s whose corresponding state in FSM' is the initial state of FSM'.

Step 7 For each state s in FSM, use perturbation operator P9 (change behavior) to match the behavior associated with s to the same behavior that is associated with its corresponding state in FSM'.

For each transition t in FSM, use perturbation operator P8 (change condition) to match the condition associated with t to the same condition that is associated with its corresponding transition in FSM'.

Step 8 For each state s in FSM, use perturbation operator P11 (change behavior parameters) to match the behavioral parameters of the behavior associated with s to the same parameters as of the behavior that is associated with the corresponding state of s in FSM'.

For each transition t in FSM, use perturbation operator P10 (change condition parameters) to match the conditional parameters of the condition associated with t to the same parameters as of the condition that is associated with the corresponding transition of t in FSM'.

□

Behavior trees

Completeness of perturbation operators

In this section, I provide proofs that any valid behavior tree can be transformed into any other valid behavior tree through the application of the perturbation operators P1 - P7 (see Section A.3).

Let BT and BT' be two behavior trees that are valid instances of control software according to the previous definition. BT can be transformed into BT' through the use of the perturbation operators P1 - P7.

Proof. The proof of the aforementioned statement will be divided into several steps. After all steps have been executed, the initial behavior tree BT has been transformed into BT'.

By definition, BT and BT' already contain the same single root (V1) and the same single top-level node that is, by definition, a sequence* ($\rightarrow^*$) node (V2). The following steps will ensure that all other parts of the behavior tree will be identical as well:

1. add selector subtrees, if necessary;
2. remove selector subtrees, if necessary;
3. update modules;
4. update parameters.

Step 1 If BT has fewer selector subtrees than BT', then apply the perturbation operator P1 (add subtree) repeatedly until the number of selector subtrees is the same in the two trees.

Step 2 If BT has more selector subtrees than BT', then apply the perturbation operator P2 (remove subtree) repeatedly until the number of selector subtrees is the same in the two trees. Step 1 and step 2 ensure that the two trees are structurally the same and all inner nodes already have the correct control node type associated. Additionally, create a matching between each leaf node in BT to a leaf node in BT', matching the two positionally identical leaf nodes in each behavior tree with each other.

Step 3 Traverse the leaf nodes of BT. If the current leaf node is a condition node, apply perturbation operator P4 (change condition) changing the condition to the one in the corresponding condition node in BT'. If the leaf node is an action node, apply perturbation operator P5 (change behavior) to change the behavior associated with that action node to the one associated with the corresponding node in BT'.

Step 4 Traverse the leaf nodes of BT. If the current leaf node is a condition node, apply the perturbation operator P6 (change condition parameter) onto that node matching the parameters of the corresponding node in BT'. If the leaf node is an action node, apply the perturbation operator P7 (change behavior parameter) onto that node to match the parameters of the corresponding node in BT'. □

Bibliography

Aarts, E., Korst, J., and Michiels, W. (2005). "Simulated annealing". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by E. K. Burke and G. Kendall. Boston, MA, USA: Springer, pp. 187–210. DOI: 10.1007/0-387-28356-0_7.

Abbeel, P. and Ng, A. Y. (2004). "Apprenticeship learning via inverse reinforcement learning". In: *ICML 2004*. Ed. by C. Brodley. New York, NY, USA: ACM, p. 1. DOI: 10.1145/1015330.1015430.

Alharthi, K., Abdallah, Z. S., and Hauert, S. (2022). "Understandable controller extraction from video observations of swarms". In: *Swarm Intelligence: 13th International Conference, ANTS 2022*. Ed. by M. Dorigo, H. Hamann, M. López-Ibáñez, J. García-Nieto, A. Engelbrecht, C. Pinciroli, V. Strobel, and C. Camacho-Villalón. Vol. 13491. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 41–53. DOI: 10.1007/978-3-031-20176-9_4.

Allen, J. M., Joyce, R., Millard, A. G., and Gray, I. (2020). "The Pi-puck ecosystem: hardware and software support for the e-puck and e-puck2". In: *Swarm Intelligence: 12th International Conference, ANTS 2020*. Ed. by M. Dorigo, T. Stützle, M. J. Blesa, C. Blum, H. Hamann, M. K. Heinrich, and V. Strobel. Vol. 12421. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 243–255. DOI: 10.1007/978-3-030-60376-2_19.

Ansótegui, C., Sellmann, M., and Tierney, K. (2009). "A gender-based genetic algorithm for the automatic configuration of algorithms". In: *Principles and Practice of Constraint Programming - CP 2009*. Ed. by I. P. Gent. Vol. 5732. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 142–157. DOI: 10.1007/978-3-642-04244-7_14.

Aristotle (1967). *Politics*. Translated by H. Rackham. Cambridge, MA, USA: Harvard University Press.

Arvin, F., Turgut, A. E., Bazyari, F., Arikan, K. B., Bellotto, N., and Yue, S. (2014). "Cue-based aggregation with a mobile robot swarm: a novel fuzzy-based method". In: *Adaptive Behavior* 22.3, pp. 189–206. DOI: 10.1177/1059712314528009.

Bäck, T., Fogel, D. B., and Michalewicz, Z., eds. (1997). *Handbook of Evolutionary Computation*. First. Bristol, United Kingdom: IOP Publishing.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). "Improvement strategies for the F-Race algorithm: sampling design and iterative refinement". In: *Hybrid Metaheuristics: 4th International Workshop, HM 2007*. Ed. by T. Bartz-Beielstein, M. J. Blesa, C. Blum, B. Naujoks, A. Roli, G. Rudolph, and M. Sampels. Vol. 4771. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 108–122. DOI: 10.1007/978-3-540-75514-2_9.

Beni, G. (2005). "From swarm intelligence to swarm robotics". In: *Swarm Robotics: SAB 2004 International Workshop*. Ed. by E. Şahin and W. M. Spears. Vol. 3342. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 1–9. DOI: 10.1007/978-3-540-30552-1_1.

Berlinger, F., Gauci, M., and Nagpal, R. (2021). "Implicit coordination for 3D underwater collective behaviors in a fish-inspired robot swarm". In: *Science Robotics* 6.50, eabd8668. DOI: 10.1126/scirobotics.abd8668.

Berman, S., Halász, Á. M., Hsieh, M. A., and Kumar, V. (2009). "Optimized stochastic policies for task allocation in swarms of robots". In: *IEEE Transactions on Robotics* 25.4, pp. 927–937. DOI: 10.1109/TRO.2009.2024997.

Berman, S., Kumar, V., and Nagpal, R. (2011). "Design of control policies for spatially inhomogeneous robot swarms with application to commercial pollination". In: *2011 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 378–385. DOI: 10.1109/ICRA.2011.5980440.

Bianco, R. and Nolfi, S. (2004). "Toward open-ended evolutionary robotics: evolving elementary robotic units able to self-assemble and self-reproduce". In: *Connection Science* 16.4, pp. 227–248. DOI: 10.1080/09540090412331314759.

Birattari, M. (2004). "The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective". PhD thesis. Brussels, Belgium: Université Libre de Bruxelles.

Birattari, M., Ligot, A., Bozhinoski, D., Brambilla, M., Francesca, G., Garattoni, L., Garzón Ramos, D., Hasselmann, K., Kegeleirs, M., Kuckling, J., Pagnozzi, F., Roli, A., Salman, M., and Stützle, T. (2019). "Automatic off-line design of robot swarms: a manifesto". In: *Frontiers in Robotics and AI* 6, p. 59. DOI: 10.3389/frobt.2019.00059.

Birattari, M., Ligot, A., and Hasselmann, K. (2020). "Disentangling automatic and semi-automatic approaches to the optimization-based design of control software for robot swarms". In: *Nature Machine Intelligence* 2.9, pp. 494–499. DOI: 10.1038/s42256-020-0215-0.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). "A racing algorithm for configuring metaheuristics". In: *GECCO'02: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Ed. by W. B.

Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. K. Burke, and N. Jonoska. San Francisco, CA, USA: Morgan Kaufmann Publishers, pp. 11–18.

Bloom, J., Mukherjee, A., and Pinciroli, C. (2022). *A study of reinforcement learning algorithms for aggregates of minimalistic robots*. https://arxiv.org/abs/2203.15129.

Bongard, J. C. and Lipson, H. (2004). "Once more unto the breach: co-evolving a robot and its simulator". In: *Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems*. Ed. by J. B. Pollack, M. A. Bedau, P. Husbands, R. A. Watson, and T. Ikegami. A Bradford Book. Cambridge, MA, USA: MIT Press, pp. 57–62. DOI: 10.7551/mitpress/1429.003.0011.

Boston Dynamics (2017). *What's new, Atlas?* https://youtu.be/fRj34o4hN4I. accessed on 2022-12-23.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, United Kingdom: Cambridge University Press. DOI: 10.1017/CBO9780511804441.

Bozhinoski, D. and Birattari, M. (2018). "Designing control software for robot swarms: software engineering for the development of automatic design methods". In: *RoSE'18: Proceedings of the 1st International Workshop on Robotics Software Engineering*. New York, NY, USA: ACM, pp. 33–35. DOI: 10.1145/3196558.3196564.

Bozhinoski, D. and Birattari, M. (2022). "Towards an integrated automatic design process for robot swarms". In: *Open Research Europe* 1, p. 112. DOI: 10.12688/openreseurope.14025.1.

Brambilla, M., Brutschy, A., Dorigo, M., and Birattari, M. (2014). "Property-driven design for swarm robotics: a design method based on prescriptive modeling and model checking". In: *ACM Transactions on Autonomous Adaptive Systems* 9.4, 17:1–17:28. DOI: 10.1145/2700318.

Brambilla, M., Ferrante, E., Birattari, M., and Dorigo, M. (2013). "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1, pp. 1–41. DOI: 10.1007/s11721-012-0075-2.

Bredeche, N. and Fontbonne, N. (2021). "Social learning in swarm robotics". In: *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 377.1843, p. 20200309. DOI: 10.1098/rstb.2020.0309.

Bredeche, N., Haasdijk, E., and Prieto, A. (2018). "Embodied evolution in collective robotics: a review". In: *Frontiers in Robotics and AI* 5, p. 12. DOI: 10.3389/frobt.2018.00012.

Bredeche, N., Montanier, J.-M., Liu, W., and Winfield, A. F. T. (2012). "Environment-driven distributed evolutionary adaptation in a population of autonomous

robotic agents". In: *Mathematical and Computer Modelling of Dynamical Systems* 18.1, pp. 101–129. DOI: 10.1080/13873954.2011.601425.

Brooks, R. A. (1992). "Artificial life and real robots". In: *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Ed. by F. J. Varela and P. Bourgine. Cambridge, MA, USA: MIT Press, pp. 3–10.

Burke, E. K. and Bykov, Y. (2017). "The late acceptance hill-climbing heuristic". In: *European Journal of Operational Research* 258.1, pp. 70–78. DOI: 10.1016/j.ejor.2016.07.012.

Cambier, N., Albani, D., Frémont, V., Trianni, V., and Ferrante, E. (2021). "Cultural evolution of probabilistic aggregation in synthetic swarms". In: *Applied Soft Computing* 113.B, p. 108010. DOI: 10.1016/j.asoc.2021.108010.

Cambier, N. and Ferrante, E. (2022). "AutoMoDe-Pomodoro: an evolutionary class of modular designs". In: *GECCO'22: Proceedings of the Genetic and Evolutionary Computation Conference*. Ed. by J. E. Fieldsend. New York, NY, USA: ACM, pp. 100–103. DOI: 10.1145/3520304.3529031.

Campo, A., Garnier, S., Dédriche, O., Zekkri, M., and Dorigo, M. (2011). "Self-organized discrimination of resources". In: *PLOS ONE* 6.5, e19888. DOI: 10.1371/journal.pone.0019888.

Campo, A., Gutiérrez, Á., Nouyan, S., Pinciroli, C., Longchamp, V., Garnier, S., and Dorigo, M. (2010). "Artificial pheromone for path selection by a foraging swarm of robots". In: *Biological Cybernetics* 103.5, pp. 339–352. DOI: 10.1007/s00422-010-0402-x.

Castelló Ferrer, E., Yamamoto, T., Dalla Libera, F., Liu, W., Winfield, A. F. T., Nakamura, Y., and Ishiguro, H. (2016). "Adaptive foraging for simulated and real robotic swarms: the dynamical response threshold approach". In: *Swarm Intelligence* 10.1, pp. 1–31. DOI: 10.1007/s11721-015-0117-7.

Chambers, J. M., Cleveland, W. S., Kleiner, B., and Tukey, P. A. (1983). *Graphical Methods For Data Analysis*. Belmont, CA, USA: CRC Press.

Champandard, A. J., Dawe, M., and Hernandez-Cerpa, D. (2010). *Behavior trees: three ways of cultivating game AI*. https://www.gdcvault.com/play/1012744/Behavior-Trees-Three-Ways-of. Game Developers Conference, AI Summit.

Cheung, N. (2017). *Technology Behind the Intel Drone Light Shows*. https://www.roboticstomorrow.com/article/2017/05/technology-behind-the-intel-drone-light-shows/10022/. accessed on 2023-01-07.

Chignoli, M., Kim, D., Stanger-Jones, E., and Kim, S. (2021). "The MIT humanoid robot: design, motion planning, and control for acrobatic behaviors". In: *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*.

Ed. by T. Asfour. Piscataway, NJ, USA: IEEE, pp. 1–8. DOI: 10.1109/HUMANOIDS47582.2021.9555782.

Christensen, A. L. and Dorigo, M. (2006). "Evolving an integrated phototaxis and hole-avoidance behavior for a swarm-bot". In: *Artificial Life X: Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems*. Ed. by L. M. Rocha, L. S. Yaeger, M. A. Bedau, D. Floreano, R. L. Goldstone, and A. Vespignani. A Bradford Book. Cambridge, MA, USA: MIT Press, pp. 248–254.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019). "Quantifying generalization in reinforcement learning". In: *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 1282–1289. DOI: 10.1007/978-3-540-71541-2_4.

Colledanchise, M. and Ögren, P. (2018). *Behavior Trees in Robotics and AI: An Introduction*. First. Chapman & Hall/CRC Artificial Intelligence and Robotics Series. Boca Raton, FL, USA: CRC Press. DOI: 10.1201/9780429489105.

Conover, W. J. (1999). *Practical Nonparametric Statistics*. Third. Wiley Series in Probability and Statistics. New York, NY, USA: John Wiley & Sons.

Correll, N. and Martinoli, A. (2007). "Robust distributed coverage using a swarm of miniature robots". In: *2007 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 379–384. DOI: 10.1109/ROBOT.2007.363816.

Dantzig, G. B. (1990). "Origins of the simplex method". In: *A history of scientific computing*. Ed. by S. G. Nash. New York, NY, USA: ACM, pp. 141–151. DOI: 10.1145/87252.88081.

De Masi, G., Prasetyo, J., Mankovskii, N., Ferrante, E., and Tuci, E. (2021). "Robot swarm democracy: the importance of informed individuals against zealots". In: *Swarm Intelligence* 15.4, pp. 315–338. DOI: 10.1007/s11721-021-00197-3.

Diggelen, F. van, Luo, J., Karagüzel, T. A., Cambier, N., Ferrante, E., and Eiben, A. (2022). "Environment induced emergence of collective behavior in evolving swarms with limited sensing". In: *GECCO'22: Proceedings of the Genetic and Evolutionary Computation Conference*. Ed. by J. E. Fieldsend. New York, NY, USA: ACM, pp. 31–39. DOI: 10.1145/3512290.3528735.

Dimidov, C., Oriolo, G., and Trianni, V. (2016). "Random walks in swarm robotics: an experiment with Kilobots". In: *Swarm Intelligence: 10th International Conference, ANTS 2016*. Ed. by M. Dorigo, M. Birattari, X. Li, M. López-Ibáñez, K. Ohkura, T. Stützle, and C. Pinciroli. Vol. 9882. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 185–196. DOI: 10.1007/978-3-319-44427-7_16.

Divband Soorati, M. and Hamann, H. (2015). "The effect of fitness function design on performance in evolutionary robotics: the influence of a priori knowledge". In: *GECCO'15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. Ed. by S. Silva. New York, NY, USA: ACM, pp. 153–160. DOI: 10.1145/2739480.2754676.

Dorigo, M., Birattari, M., and Brambilla, M. (2014). "Swarm robotics". In: *Scholarpedia* 9.1, p. 1463. DOI: 10.4249/scholarpedia.1463.

Dorigo, M., Floreano, D., Gambardella, L. M., Mondada, F., Nolfi, S., Baaboura, T., Birattari, M., Bonani, M., Brambilla, M., Brutschy, A., Burnier, D., Campo, A., Christensen, A. L., Decugnière, A., Di Caro, G. A., Ducatelle, F., Ferrante, E., Förster, A., Martinez Gonzales, J., Guzzi, J., Longchamp, V., Magnenat, S., Mathews, N., Montes de Oca, M., O'Grady, R., Pinciroli, C., Pini, G., Retornaz, P., Roberts, J., Sperati, V., Stirling, T., Stranieri, A., Stützle, T., Trianni, V., Tuci, E., Turgut, A. E., and Vaussard, F. (2013). "Swarmanoid: a novel concept for the study of heterogeneous robotic swarms". In: *IEEE Robotics & Automation Magazine* 20.4, pp. 60–71. DOI: 10.1109/MRA.2013.2252996.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). "Ant system: optimization by a colony of cooperating agents". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1, pp. 29–41. DOI: 10.1109/3477.484436.

Dorigo, M. and Stützle, T. (2014). *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press.

Dorigo, M., Theraulaz, G., and Trianni, V. (2020). "Reflections on the future of swarm robotics". In: *Science Robotics* 5, eabe4385. DOI: 10.1126/scirobotics.abe4385.

Dorigo, M., Theraulaz, G., and Trianni, V. (2021). "Swarm robotics: past, present, and future [point of view]". In: *Proceedings of the IEEE* 109.7, pp. 1152–1165. DOI: 10.1109/JPROC.2021.3072740.

Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella Thomas, H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D., and Gambardella, L. M. (2003). "Evolving self-organizing behaviors for a Swarm-bot". In: *Autonomous Robots* 17, pp. 223–245. DOI: 10.1023/B:AURO.0000033973.24945.f3.

Dosieah, G. Y., Özdemir, A., Gauci, M., and Groß, R. (2022). "Moving mixtures of active and passive elements with robots that do not compute". In: *Swarm Intelligence: 13th International Conference, ANTS 2022*. Vol. 13491. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 183–195. DOI: 10.1007/978-3-031-20176-9_15.

Duarte, M., Costa, V., Gomes, J., Rodrigues, T., Silva, F., Oliveira, S. M., and Christensen, A. L. (2016). "Evolution of collective behaviors for a real swarm

of aquatic surface robots". In: *PLOS ONE* 11.3, e0151834. DOI: 10.1371/journal.pone.0151834.

Duarte, M., Oliveira, S. M., and Christensen, A. L. (2014). "Hybrid control for large swarms of aquatic drones". In: *ALIFE 14: The Fourteenth International Conference on the Synthesis and Simulation of Living Systems*. Ed. by H. Sayama, J. Rieffel, S. Risi, R. Doursat, and H. Lipson. Cambridge, MA, USA: MIT Press, pp. 785–792. DOI: 10.7551/978-0-262-32621-6-ch105.

Duarte, M., Oliveira, S. M., and Christensen, A. L. (2015). "Evolution of hybrid robotic controllers for complex tasks". In: *Journal of Intelligent & Robotic Systems* 78.3-4, pp. 463–484. DOI: 10.1007/s10846-014-0086-x.

Eberhart, R. and Kennedy, J. (1995). "A new optimizer using particle swarm theory". In: *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, MHS'95*. Piscataway, NJ, USA: IEEE, pp. 39–43. DOI: 10.1109/MHS.1995.494215.

Ebert, J. T., Gauci, M., and Nagpal, R. (2017). "Multi-feature collective decision making in robot swarms". In: *AAMAS '18: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1711–1719.

École polytechnique fédérale de Lausanne (2010). *Omnidirectional vision turret for the e-puck*. http://www.e-puck.org/index.php?option=com_content&view=article&id=26&Itemid=21.

Edmonds, J. and Karp, R. M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM* 19.2, pp. 248–264. DOI: 10.1145/321694.321699.

Elamvazhuthi, K. and Berman, S. (2019). "Mean-field models in swarm robotics: a survey". In: *Bioinspiration & Biomimetics* 15, p. 015001. DOI: 10.1088/1748-3190/ab49a4.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). "Neural architecture search: a survey". In: *Journal of Machine Learning Research* 20.55, pp. 1–21.

Engebråten, S. A., Moen, J., Yakimenko, O. A., and Glette, K. (2020). "A framework for automatic behavior generation in multi-function swarms". In: *Frontiers in Robotics and AI* 7, p. 175. DOI: 10.3389/frobt.2020.579403.

Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., and O'Neill, M. (2017). *PonyGE2: grammatical evolution in Python*. https://arxiv.org/abs/1703.08535.

Feo, T. A. and Resende, M. G. C. (1989). "A probabilistic heuristic for a computationally difficult set covering problem". In: *Operations Research Letters* 8.2, pp. 67–71. DOI: 10.1016/0167-6377(89)90002-3.

Ferrante, E., Duéñez-Guzmán, E. A., Turgut, A. E., and Wenseleers, T. (2013). "GESwarm: grammatical evolution for the automatic synthesis of collective

behaviors in swarm robotics". In: *GECCO'13: Proceedings of the 15th annual conference on Genetic and evolutionary computation*. Ed. by C. Blum. New York, NY, USA: ACM, pp. 17–24. DOI: 10.1145/2463372.2463385.

Ferrante, E., Turgut, A. E., Duéñez-Guzmán, E. A., Dorigo, M., and Wenseleers, T. (2015). "Evolution of self-organized task specialization in robot swarms". In: *PLOS Computational Biology* 11.8, e1004273. DOI: 10.1371/journal.pcbi.1004273.

Feurer, M. and Hutter, F. (2019). "Hyperparameter optimization". In: *Automated Machine Learning*. Ed. by F. Hutter, L. Kotthoff, and J. Vanschoren. The Springer Series on Challenges in Machine Learning. Cham, Switzerland: Springer, pp. 3–33. DOI: 10.1007/978-3-030-05318-5_1.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). "DEAP: evolutionary algorithms made easy". In: *Journal of Machine Learning Research* 13, pp. 2171–2175.

Francesca, G. (2017). "A modular approach to the automatic design of control software for robot swarms: from a novel perspective on the reality gap to AutoMoDe". PhD thesis. Brussels, Belgium: Université Libre de Bruxelles.

Francesca, G. and Birattari, M. (2016). "Automatic design of robot swarms: achievements and challenges". In: *Frontiers in Robotics and AI* 3.29, pp. 1–9. DOI: 10.3389/frobt.2016.00029.

Francesca, G., Brambilla, M., Brutschy, A., Garattoni, L., Miletitch, R., Podevijn, G., Reina, A., Soleymani, T., Salvaro, M., Pinciroli, C., Mascia, F., Trianni, V., and Birattari, M. (2015). "AutoMoDe-Chocolate: automatic design of control software for robot swarms". In: *Swarm Intelligence* 9.2–3, pp. 125–152. DOI: 10.1007/s11721-015-0107-9.

Francesca, G., Brambilla, M., Brutschy, A., Trianni, V., and Birattari, M. (2014). "AutoMoDe: a novel approach to the automatic design of control software for robot swarms". In: *Swarm Intelligence* 8.2, pp. 89–112. DOI: 10.1007/s11721-014-0092-4.

Francesca, G., Brambilla, M., Trianni, V., Dorigo, M., and Birattari, M. (2012). "Analysing an evolved robotic behaviour using a biological model of collegial decision making". In: *From Animals to Animats 12: 12th International Conference on Simulation of Adaptive Behavior, SAB 2012*. Ed. by T. Ziemke, C. Balkenius, and J. Hallam. Vol. 7426. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 381–390. DOI: 10.1007/978-3-642-33093-3_38.

Franzin, A. and Stützle, T. (2019). "Revisiting simulated annealing: a component-based analysis". In: *Computers & Operations Research* 104, pp. 191–206. DOI: 10.1016/j.cor.2018.12.015.

Friedman, M. (1937). “The use of ranks to avoid the assumption of normality implicit in the analysis of variance”. In: *Journal of the American Statistical Association* 32.200, pp. 675–701. DOI: 10.1080/01621459.1937.10503522.

Friedman, M. (1939). “A correction: the use of ranks to avoid the assumption of normality implicit in the analysis of variance”. In: *Journal of the American Statistical Association* 34.205, p. 109. DOI: 10.1080/01621459.1939.10502372.

Friston, K. (2010). “The free-energy principle: a unified brain theory?” In: *Nature Reviews Neuroscience* 11, pp. 127–138. DOI: 10.1038/nrn2787.

Garattoni, L. and Birattari, M. (2018). “Autonomous task sequencing in a robot swarm”. In: *Science Robotics* 3.20, eaat0430. DOI: 10.1126/scirobotics.aat0430.

Garattoni, L., Francesca, G., Brutschy, A., Pinciroli, C., and Birattari, M. (2015). *Software infrastructure for e-puck (and TAM)*. Tech. rep. TR/IRIDIA/2015-004. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Garnier, S., Jost, C., Jeanson, R., Gautrais, J., Asadpour, M., Caprari, G., and Theraulaz, G. (2005). “Aggregation behaviour as a source of collective decision in a group of cockroach-like-robots”. In: *Advances in Artificial Life: 8th European Conference, ECAL 2005*. Ed. by M. S. Capcarrère, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis. Vol. 3630. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 169–178. DOI: 10.1007/11553090_18.

Garzón Ramos, D. and Birattari, M. (2020). “Automatic design of collective behaviors for robots that can display and perceive colors”. In: *Applied Sciences* 10.13, p. 4654. DOI: 10.3390/app10134654.

Gauci, M., Chen, J., Dodd, T. J., and Groß, R. (2014a). “Evolving aggregation behaviors in multi-robot systems with binary sensors”. In: *Distributed Autonomous Robotic Systems: The 11th International Symposium*. Ed. by M. A. Hsieh and G. Chirikjian. Vol. 104. Springer Tracts in Advanced Robotics. Berlin, Germany: Springer, pp. 355–367. DOI: 10.1007/978-3-642-55146-8_25.

Gauci, M., Chen, J., Li, W., Dodd, T. J., and Groß, R. (2014b). “Clustering objects with robots that do not compute”. In: *AAMAS ’14: Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 421–428.

Gauci, M., Chen, J., Li, W., Dodd, T. J., and Groß, R. (2014c). “Self-organized aggregation without computation”. In: *The International Journal of Robotics Research* 33.8, pp. 1145–1161. DOI: 10.1177/0278364914525244.

Gazi, V. (2005). “Swarm aggregations using artificial potentials and sliding-mode control”. In: *IEEE Transactions on Robotics* 21.6, pp. 1208–1214. DOI: 10.1109/TRO.2005.853487.

Geman, S., Bienenstock, E., and Doursat, R. (1992). "Neural networks and the bias/variance dilemma". In: *Neural Computation* 4.1, pp. 1–58. DOI: 10.1162/neco.1992.4.1.1.

Gharbi, I., Kuckling, J., Garzón Ramos, D., and Birattari, M. (2023). "Show me what you want: inverse reinforcement learning to automatically design robot swarms by demonstration". In: submitted for publication.

Giernacki, W., Skwierczyński, M., Witwicki, W., Wroński, P., and Kozierski, P. (2017). "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering". In: *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*. Piscataway, NJ, USA: IEEE, pp. 37–42. DOI: 10.1109/MMAR.2017.8046794.

Glasmachers, T., Schaul, T., Yi, S., Wierstra, D., and Schmidhuber, J. (2010). "Exponential natural evolution strategies". In: *GECCO'10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, pp. 393–400. DOI: 10.1145/1830483.1830557.

Glover, F. (1989). "Tabu search–part I". In: *ORSA Journal on Computing* 1.3, pp. 190–206. DOI: 10.1287/ijoc.1.3.190.

Glover, F. and Kochenberger, G. A., eds. (2003). *Handbook of Metaheuristics*. Vol. 57. International Series in Operations Research & Management Science. Boston, MA, USA: Springer. DOI: 10.1007/b101874.

Gomes, J. and Christensen, A. L. (2018). "Task-agnostic evolution of diverse repertoires of swarm behaviours". In: *Swarm Intelligence: 11th International Conference, ANTS 2018*. Ed. by M. Dorigo, M. Birattari, C. Blum, A. L. Christensen, A. Reina, and V. Trianni. Vol. 11172. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 225–238. DOI: 10.1007/978-3-030-00533-7_18.

Gomes, J., Mariano, P., and Christensen, A. L. (2017). "Novelty-driven cooperative coevolution". In: *Evolutionary Computation* 25.2, pp. 257–307. DOI: 10.1162/EVCO_a_00173.

Gomes, J., Mariano, P., and Christensen, A. L. (2019). "Challenges in cooperative coevolution of physically heterogeneous robot teams". In: *Natural Computing* 18, pp. 29–46. DOI: 10.1007/s11047-016-9582-1.

Gomes, J., Urbano, P., and Christensen, A. L. (2013). "Evolution of swarm robotics systems with novelty search". In: *Swarm Intelligence* 7.2–3, pp. 115–144. DOI: 10.1007/s11721-013-0081-z.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. First. Cambridge, MA, USA: MIT Press.

Gutiérrez, Á., Campo, A., Dorigo, M., Donate, J., Monasterio-Huelin, F., and Magdalena, L. (2009). "Open e-puck range & bearing miniaturized board for local communication in swarm robotics". In: *2009 IEEE International*

Conference on Robotics and Automation (ICRA). Ed. by K. Kosuge. Piscataway, NJ, USA: IEEE, pp. 3111–3116. DOI: 10.1109/ROBOT.2009.5152456.

Hajek, B. (1988). "Cooling schedules for optimal annealing". In: *Mathematics of Operations Research* 13.2, pp. 311–329. DOI: 10.1287/moor.13.2.311.

Halloy, J., Sempo, G., Caprari, G., Rivault, C., Asadpour, M., Tâche, F., Saïd, I., Durier, V., Canonge, S., Amé, J.-M., Detrain, C., Correll, N., Martinoli, A., Mondada, F., Siegwart, R., and Deneubourg, J.-L. (2007). "Social integration of robots into groups of cockroaches to control self-organized choices". In: *Science* 318.5853, pp. 1155–1158. DOI: 10.1126/science.1144259.

Hamann, H. (2014). "Evolution of collective behaviors by minimizing surprise". In: *ALIFE 14: The Fourteenth International Conference on the Synthesis and Simulation of Living Systems*. Ed. by H. Sayama, J. Rieffel, S. Risi, R. Doursat, and H. Lipson. Cambridge, MA, USA: MIT Press, pp. 344–351. DOI: 10.1162/978-0-262-32621-6-ch055.

Hamann, H. (2018). *Swarm robotics: a formal approach*. Cham, Switzerland: Springer. ISBN: 978-3-319-74526-8. DOI: 10.1007/978-3-319-74528-2.

Hamann, H., Schmickl, T., Wörn, H., and Crailsheim, K. (2012). "Analysis of emergent symmetry breaking in collective decision making". In: *Neural Computing and Applications* 21.2, pp. 207–218. DOI: 10.1007/s00521-010-0368-6.

Hamann, H., Valentini, G., Khaluf, Y., and Dorigo, M. (2014). "Derivation of a micro-macro link for collective decision-making systems". In: *Parallel Problem Solving from Nature – PPSN XIII: 13th International Conference*. Ed. by T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith. Vol. 8672. Lecture Notes in Computer Science 1. Cham, Switzerland: Springer, pp. 181–190. DOI: 10.1007/978-3-319-10762-2.

Hamann, H. and Wörn, H. (2008). "A framework of space–time continuous models for algorithm design in swarm robotics". In: *Swarm Intelligence* 2.2–4, pp. 209–239. DOI: 10.1007/s11721-008-0015-3.

Hamann, H., Wörn, H., Crailsheim, K., and Schmickl, T. (2008). "Spatial macroscopic models of a bio-inspired robotic swarm algorithm". In: *2008 IEEE/RSJ International Conference On Intelligent Robots And Systems (IROS)*. Piscataway, NJ, USA: IEEE, pp. 1415–1420. DOI: 10.1109/IROS.2008.4651038.

Hannaford, B., Rosen, J., Friedman, D. C. W., King, H., Roan, P., Cheng, L., Glozman, D., Ma, J., Nia Kosari, S., and White, L. (2013). "Raven-II: an open platform for surgical robotics research". In: *IEEE Transactions on Biomedical Engineering* 60.4, pp. 954–959. DOI: 10.1109/TBME.2012.2228858.

Hansen, N. and Ostermeier, A. (2001). "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary Computation* 9.2, pp. 159–195. DOI: 10.1162/106365601750190398.

Hasan, A. (2022). "Building an integrated framework for the automatic modular design of robot swarms". MA thesis. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Hasselmann, K. (2023). "Advances in the automatic modular design of control software for robot swarms: Using neuroevolution to generate modules". PhD thesis. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Hasselmann, K. and Birattari, M. (2020). "Modular automatic design of collective behaviors for robots endowed with local communication capabilities". In: *PeerJ Computer Science* 6, e291. DOI: 10.7717/peerj-cs.291.

Hasselmann, K., Ligot, A., and Birattari, M. (2023). "Towards the automatic design of automatic methods for the design of robot swarms". In: submitted for journal publication.

Hasselmann, K., Ligot, A., Francesca, G., Garzón Ramos, D., Salman, M., Kuckling, J., Mendiburu, F. J., and Birattari, M. (2018). *Reference models for AutoMoDe*. Tech. rep. TR/IRIDIA/2018-002. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Hasselmann, K., Ligot, A., Ruddick, J., and Birattari, M. (2021). "Empirical assessment and comparison of neuro-evolutionary methods for the automatic off-line design of robot swarms". In: *Nature Communications* 12, p. 4345. DOI: 10.1038/s41467-021-24642-3.

Hecker, J. P., Letendre, K., Stolleis, K., Washington, D., and Moses, M. E. (2012). "Formica ex machina: ant swarm foraging from physical to virtual and back again". In: *Swarm Intelligence: 8th International Conference, ANTS 2012*. Ed. by M. Dorigo, M. Birattari, C. Blum, A. L. Christensen, A. P. Engelbrecht, R. Groß, and T. Stützle. Vol. 7461. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 252–259. DOI: 10.1007/978-3-642-32650-9_25.

Heinerman, J., Rango, M., and Eiben, A. (2015). "Evolution, individual learning, and social learning in a swarm of real robots". In: *2015 IEEE Symposium Series on Computational Intelligence, SSCI 2015*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1055–1062. DOI: 10.1109/SSCI.2015.152.

Hoff, N. R., Sagoff, A., Wood, R. J., and Nagpal, R. (2010). "Two foraging algorithms for robot swarms using only local communication". In: *2010 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Piscataway, NJ, USA: IEEE, pp. 123–130. DOI: 10.1109/ROBIO.2010.5723314.

Hoos, H. H. and Stützle, T. (2005). *Stochastic Local Search: Foundations & Applications*. First. San Francisco, CA, USA: Morgan Kaufmann Publishers. DOI: 10.1016/B978-1-55860-872-6.X5016-1.

Hu, D., Gong, Y., Hannaford, B., and Seibel, E. J. (2015). "Semi-autonomous simulated brain tumor ablation with RavenII surgical robot using behavior tree". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 3868–3875. DOI: 10.1109/ICRA.2015.7139738.

Hüttenrauch, M., Šošić, A., and Neumann, G. (2019). "Deep reinforcement learning for swarm systems". In: *Journal of Machine Learning Research* 20.54, pp. 1–31.

Hutter, F., Hoos, H. H., and Leyton Brown, K. (2011). "Sequential model-based optimization for general algorithm configuration". In: *Learning and Intelligent Optimization: 5th International Conference, LION 5*. Ed. by C. A. Coello Coello. Vol. 6683. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40.

Hutter, F., Hoos, H. H., Leyton Brown, K., and Stützle, T. (2009). "ParamILS: an automatic algorithm configuration framework". In: *Journal of Artificial Intelligence Research* 36, pp. 267–306. DOI: 10.1613/jair.2861.

Ijspeert, A. J., Martinoli, A., Billard, A., and Gambardella, L. M. (2001). "Collaboration through the exploitation of local interactions in autonomous collective robotics: the stick pulling experiment". In: *Autonomous Robots* 11.2, pp. 149–171. DOI: 10.1023/A:1011227210047.

Isla, D. (2005). "Handling complexity in the Halo 2 AI". In: *Game Developers Conference, GDC 2005*. Vol. 12. London, United Kingdom: Game Developers Conference (GDC).

Jakobi, N., Husbands, P., and Harvey, I. (1995). "Noise and the reality gap: the use of simulation in evolutionary robotics". In: *Advances in Artificial Life: Third European Conference on Artificial Life*. Ed. by F. Morán, A. Moreno, J. J. Merelo, and P. Chacón. Vol. 929. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer, pp. 704–720. DOI: 10.1007/3-540-59496-5_337.

Jones, S., Milner, E., Sooriyabandara, M., and Hauert, S. (2022). *DOTS: an open testbed for industrial swarm robotic solutions*. https://arxiv.org/abs/2203.13809.

Jones, S., Studley, M., Hauert, S., and Winfield, A. F. T. (2018a). "A two teraflop swarm". In: *Frontiers in Robotics and AI* 5, p. 11. DOI: 10.3389/frobt.2018.00011.

Jones, S., Studley, M., Hauert, S., and Winfield, A. F. T. (2018b). "Evolving behaviour trees for swarm robotics". In: *Distributed Autonomous Robotic Systems: The 13th International Symposium*. Ed. by R. Groß, A. Kolling, S. Berman, E. Frazzoli, A. Martinoli, F. Matsuno, and M. Gauci. Vol. 6. Springer Proceedings in Advanced Robotics. Cham, Switzerland: Springer, pp. 487–501. DOI: 10.1007/978-3-319-73008-0_34.

Jones, S., Winfield, A. F. T., Hauert, S., and Studley, M. (2019). "Onboard evolution of understandable swarm behaviors". In: *Advanced Intelligent Systems* 1.6, p. 1900031. DOI: 10.1002/aisy.201900031.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). "Reinforcement learning: a survey". In: *Journal of Artificial Intelligence Research* 4, pp. 237–285. DOI: 10.1613/jair.301.

Kaiser, T. K. and Hamann, H. (2019). "Engineered self-organization for resilient robot self-assembly with minimal surprise". In: *Robotics and Autonomous Systems* 122, p. 103293. DOI: 10.1016/j.robot.2019.103293.

Kaiser, T. K. and Hamann, H. (2022). "Innate motivation for robot swarms by minimizing surprise: from simple simulations to real-world experiments". In: *IEEE Transactions on Robotics* 38.6, pp. 3582–3601. DOI: 10.1109/TRO.2022.3181004.

Kazadi, S. (2009). "Model independence in swarm robotics". In: *International Journal of Intelligent Computing and Cybernetics* 2.4, pp. 672–694. DOI: 10.1108/17563780911005836.

Kerschke, P., Hoos, H. H., Neumann, F., and Trautmann, H. (2019). "Automated algorithm selection: survey and perspectives". In: *Evolutionary Computation* 27.1, pp. 3–45. DOI: 10.1162/evco_a_00242.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton Brown, K. (2016). "SATenstein: automatically building local search SAT solvers from components". In: *Artificial Intelligence* 232, pp. 20–42. DOI: 10.1016/j.artint.2015.11.002.

Kirkpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P. (1983). "Optimization by simulated annealing". In: *Science* 220.4598, pp. 671–680. DOI: 10.1126/science.220.4598.671.

Kober, J., Bagnell, J. A., and Peters, J. (2013). "Reinforcement learning in robotics: a survey". In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274. DOI: 10.1177/0278364913495721.

Kochenderfer, M. J. and Wheeler, T. A. (2019). *Algorithms for Optimization*. Cambridge, MA, USA: MIT Press.

Koos, S., Mouret, J.-B., and Doncieux, S. (2013). "The transferability approach: crossing the reality gap in evolutionary robotics". In: *IEEE Transactions on Evolutionary Computation* 17.1, pp. 122–145. DOI: 10.1109/TEVC.2012.2185849.

Korte, B. and Vygen, J. (2018). *Combinatorial Optimization: Theory and Algorithms*. Sixth. Algorithms and Combinatorics. Berlin, Germany: Springer. DOI: 10.1007/978-3-662-56039-6.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. First. A Bradford Book. Cambridge, MA, USA: MIT Press.

Krieger, M. J. B., Billeter, J.-B., and Keller, L. (2000). "Ant-like task allocation and recruitment in cooperative robots". In: *Nature* 406, pp. 992–995. DOI: 10.1038/35023164.

Kuckling, J. (2023). *Optimization in the automatic modular design of control software for robot swarms: supplementary material*. https://iridia.ulb.ac.be/supp/.

Kuckling, J., Hasselmann, K., Pelt, V. van, Kiere, C., and Birattari, M. (2021a). *AutoMoDe Editor: a visualization tool for AutoMoDe*. Tech. rep. TR/IRIDIA/2021-009. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Kuckling, J., Hoos, H. H., Stützle, T., and Birattari, M. (2023). "Comparison of different optimization algorithms in the automatic modular design of control software for robot swarms". In: to be submitted for journal publication.

Kuckling, J., Ligot, A., Bozhinoski, D., and Birattari, M. (2018a). *Search space for AutoMoDe-Chocolate and AutoMoDe-Maple*. Tech. rep. TR/IRIDIA/2018-012. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Kuckling, J., Ligot, A., Bozhinoski, D., and Birattari, M. (2018b). "Behavior trees as a control architecture in the automatic modular design of robot swarms". In: *Swarm Intelligence: 11th International Conference, ANTS 2018*. Ed. by M. Dorigo, M. Birattari, C. Blum, A. L. Christensen, A. Reina, and V. Trianni. Vol. 11172. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 30–43. DOI: 10.1007/978-3-030-00533-7_3.

Kuckling, J., Pelt, V. van, and Birattari, M. (2021b). "Automatic modular design of behavior trees for robot swarms with communication capabilities". In: *Applications of Evolutionary Computation: 24th International Conference, EvoApplications 2021*. Ed. by P. A. Castillo and J. L. Jiménez Laredo. Vol. 12694. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 130–145. DOI: 10.1007/978-3-030-72699-7_9.

Kuckling, J., Pelt, V. van, and Birattari, M. (2022). "AutoMoDe-Cedrata: automatic design of behavior trees for controlling a swarm of robots with communication capabilities". In: *SN Computer Science* 3, p. 136. DOI: 10.1007/s42979-021-00988-9.

Kuckling, J., Stützle, T., and Birattari, M. (2020a). "Iterative improvement in the automatic modular design of robot swarms". In: *PeerJ Computer Science* 6, e322. DOI: 10.7717/peerj-cs.322.

Kuckling, J., Ubeda Arriaza, K., and Birattari, M. (2020b). "AutoMoDe-IcePop: automatic modular design of control software for robot swarms using simulated annealing". In: *Artificial Intelligence and Machine Learning: BNAIC 2019, BENELEARN 2019*. Ed. by B. Bogaerts, G. Bontempi, P. Geurts, N. Harley, B. Lebichot, T. Lenaerts, and G. Louppe. Vol. 1196. Communications in Computer and Information Science. Cham, Switzerland: Springer, pp. 3–17. DOI: 10.1007/978-3-030-65154-1_1.

KUKA Group (2021). *Audi relies on technology know-how from KUKA*. https://www.kuka.com/en-be/company/press/news/2021/09/audi-auftragsmeldung-karosseriebau. accessed on 2023-01-07.

Landau, D. P. and Binder, K. (2014). *A Guide to Monte Carlo Simulations in Statistical Physics*. Fourth. Cambridge, United Kingdom: Cambridge University Press. DOI: 10.1017/CBO9781139696463.

Legarda Herranz, G., Garzón Ramos, D., Kuckling, J., Kegeleirs, M., and Birattari, M. (2022). *Tycho: a robust, ROS-based tracking system for robot swarms*. Tech. rep. TR/IRIDIA/2022-009. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Lehman, J. and Stanley, K. O. (2011). "Abandoning objectives: evolution through the search for novelty alone". In: *Evolutionary Computation* 19.2, pp. 189–223. DOI: 10.1162/EVCO_a_00025.

Lerman, K. and Galstyan, A. (2002). "Mathematical model of foraging in a group of robots: effect of interference". In: *Autonomous Robots* 13.2, pp. 127–141. DOI: 10.1023/A:1019633424543.

Lerman, K., Galstyan, A., Martinoli, A., and Ijspeert, A. J. (2001). "A Macroscopic Analytical Model of Collaboration in Distributed Robotic Systems". In: *Artificial Life* 7.4, pp. 375–393. DOI: 10.1162/106454601317297013.

Lerman, K., Jones, C., Galstyan, A., and Matarić, M. J. (2006). "Analysis of dynamic task allocation in multi-robot systems". In: *The International Journal of Robotics Research* 25.3, pp. 225–241. DOI: 10.1177/0278364906063426.

Li, S., Batra, R., Brown, D., Chang, H.-D., Ranganathan, N., Hoberman, C., Rus, D., and Lipson, H. (2019). "Particle robotics based on statistical mechanics of loosely coupled components". In: *Nature* 567.7748, pp. 361–365. DOI: 10.1038/s41586-019-1022-9.

Li, W., Gauci, M., and Groß, R. (2016). "Turing learning: a metric-free approach to inferring behavior and its application to swarms". In: *Swarm Intelligence* 10, pp. 211–243. DOI: 10.1007/s11721-016-0126-1.

Ligot, A. (2023). "Assessing and forecasting the performance of optimization-based design methods for robot swarms: Experimental protocol & pseudo-reality predictors". PhD thesis. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Ligot, A. and Birattari, M. (2020). "Simulation-only experiments to mimic the effects of the reality gap in the automatic design of robot swarms". In: *Swarm Intelligence* 14, pp. 1–24. DOI: 10.1007/s11721-019-00175-w.

Ligot, A. and Birattari, M. (2022). "On using simulation to predict the performance of robot swarms". In: *Scientific Data* 9, p. 788. DOI: 10.1038/s41597-022-01895-1.

Ligot, A., Cotorruelo, A., Garone, E., and Birattari, M. (2022). "Towards an empirical practice in off-line fully-automatic design of robot swarms". In:

IEEE Transactions on Evolutionary Computation. DOI: 10.1109/TEVC.2022.3144848.

Ligot, A., Hasselmann, K., and Birattari, M. (2020a). "AutoMoDe-Arlequin: neural networks as behavioral modules for the automatic design of probabilistic finite state machines". In: *Swarm Intelligence: 12th International Conference, ANTS 2020*. Ed. by M. Dorigo, T. Stützle, M. J. Blesa, C. Blum, H. Hamann, M. K. Heinrich, and V. Strobel. Vol. 12421. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 109–122. DOI: 10.1007/978-3-030-60376-2_21.

Ligot, A., Hasselmann, K., Delhaisse, B., Garattoni, L., Francesca, G., and Birattari, M. (2017). *AutoMoDe, NEAT, and EvoStick: implementations for the e-puck robot in ARGoS3*. Tech. rep. TR/IRIDIA/2017-002. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Ligot, A., Kuckling, J., Bozhinoski, D., and Birattari, M. (2020b). "Automatic modular design of robot swarms using behavior trees as a control architecture". In: *PeerJ Computer Science* 6, e314. DOI: 10.7717/peerj-cs.314.

Lim, C.-U., Baumgarten, R., and Colton, S. (2010). "Evolving behaviour trees for the commercial game DEFCON". In: *Applications of Evolutionary Computation. EvoApplications 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC*. Ed. by C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcázar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preuss, J. Togelius, and G. N. Yannakakis. Vol. 6024. Lecture Notes in Computer Science. Cham, Switzerland: Springer, pp. 100–110. DOI: 10.1007/978-3-642-12239-2_11.

Lin, S. and Kernighan, B. W. (1973). "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations Research* 21.2, pp. 498–516. DOI: 10.1287/opre.21.2.498.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., and Hutter, F. (2022). "SMAC3: a versatile bayesian optimization package for hyperparameter optimization". In: *Journal of Machine Learning Research* 23.54, pp. 1–9.

Liu, W. and Winfield, A. F. T. (2010). "Modeling and optimization of adaptive foraging in swarm robotic systems". In: *The International Journal of Robotics Research* 29.14, pp. 1743–1760. DOI: 10.1177/0278364910375139.

Lopes, Y. K., Trenkwalder, S. M., Leal, A. B., Dodd, T. J., and Groß, R. (2016). "Supervisory control theory applied to swarm robotics". In: *Swarm Intelligence* 10.1, pp. 65–97. DOI: 10.1007/s11721-016-0119-0.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., and Stützle, T. (2016). "The irace package: iterated racing for automatic algorithm configuration". In: *Operations Research Perspectives* 3, pp. 43–58. DOI: 10.1016/j.orp.2016.09.002.

Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). "Iterated local search". In: *Handbook of Metaheuristics*. Ed. by F. Glover and G. A. Kochenberger. Vol. 59. International Series in Operations Research & Management Science. Boston, MA, USA: Springer, pp. 320–353. DOI: 10.1007/0-306-48056-5_11.

Ludwig, L. and Gini, M. (2006). "Robotic swarm dispersion using wireless intensity signals". In: *Distributed Autonomous Robotic Systems 7*. Ed. by M. Gini and R. Voyles. Tokyo, Japan: Springer, pp. 325–332. DOI: 10.1007/4-431-35881-1_14.

Lundy, M. and Alistair, M. (1986). "Convergence of an annealing algorithm". In: *Mathematical Programming* 34.1, pp. 111–124. DOI: 10.1007/BF01582166.

Maron, O. and Moore, A. W. (1997). "The Racing Algorithm: model selection for lazy learners". In: *Artificial Intelligence Review* 11.1–5, pp. 193–225. DOI: 10.1023/A:1006556606079.

Marshall, J. A. R., Reina, A., and Bose, T. (2019). "Multiscale modelling tool: mathematical modelling of collective behaviour without the maths". In: *PLOS ONE* 14.9, e0222906. DOI: 10.1371/journal.pone.0222906.

Martinoli, A., Easton, K., and Agassounon, W. (2004). "Modeling swarm robotic systems: a case study in collaborative distributed manipulation". In: *The International Journal of Robotics Research* 23.4-5, pp. 415–436. DOI: 10.1177/0278364904042197.

Martinoli, A., Ijspeert, A. J., and Mondada, F. (1999). "Understanding collective aggregation mechanisms: from probabilistic modelling to experiments with real robots". In: *Robotics and Autonomous Systems* 29.1, pp. 51–63. DOI: 10.1016/S0921-8890(99)00038-X.

Marzinotto, A., Colledanchise, M., Smith, C., and Ögren, P. (2014). "Towards a unified behavior trees framework for robot control". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 5420–5427. DOI: 10.1109/ICRA.2014.6907656.

Massink, M., Brambilla, M., Latella, D., Dorigo, M., and Birattari, M. (2013). "On the use of Bio-PEPA for modelling and analysing collective behaviours in swarm robotics". In: *Swarm Intelligence* 7.2-3, pp. 201–228. DOI: 10.1007/s11721-013-0079-6.

Matarić, M. J. (1997). "Reinforcement learning in the multi-robot domain". In: *Autonomous Robots* 4.1, pp. 73–83. DOI: 10.1023/A:1008819414322.

Matarić, M. J. (1998). "Using communication to reduce locality in distributed multi-agent learning". In: *Journal of Experimental & Theoretical Artificial Intelligence* 10.3, pp. 357–369. DOI: 10.1080/095281398146806.

Mathews, G. B. (1897). "On the partition of numbers". In: *Proceedings of the London Mathematical Society* 28, pp. 486–490. DOI: 10.1112/plms/s1-28.1.486.

Mathews, N., Christensen, A. L., O'Grady, R., Mondada, F., and Dorigo, M. (2017). "Mergeable nervous systems for robots". In: *Nature Communications* 8.1, p. 439. DOI: 10.1038/s41467-017-00109-2.

Matthey, L., Berman, S., and Kumar, V. (2009). "Stochastic strategies for a swarm robotic assembly system". In: *2009 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1953–1958. DOI: 10.1109/ROBOT.2009.5152457.

Mendiburu, F. J., Garzón Ramos, D., Morais, M. R. A., Lima, A. M. N., and Birattari, M. (2022). "AutoMoDe-Mate: automatic off-line design of spatially-organizing behaviors for robot swarms". In: *Swarm and Evolutionary Computation* 74, p. 101118. DOI: 10.1016/j.swevo.2022.101118.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21.6, pp. 1087–1092. DOI: 10.1063/1.1699114.

Mitra, D., Romeo, F., and Sangiovanni-Vincentelli, A. (1985). "Convergence and finite-time behavior of simulated annealing". In: *1985 24th IEEE Conference on Decision and Control*. Piscataway, NJ, USA: IEEE, pp. 761–767. DOI: 10.1109/CDC.1985.268600.

Mladenović, N. and Hansen, P. (1997). "Variable neighborhood search". In: *Computers & Operations Research* 24.11, pp. 1097–1100. DOI: 10.1016/S0305-0548(97)00031-2.

Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.-C., Floreano, D., and Martinoli, A. (2009). "The e-puck, a robot designed for education in engineering". In: *ROBOTICA 2009: Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*. Ed. by P. Gonçalves, P. Torres, and C. Alves. Castelo Branco, Portugal: Instituto Politécnico de Castelo Branco, pp. 59–65.

Mondada, F., Franzi, E., and Ienne, P. (1997). "Mobile robot miniaturisation: a tool for investigation in control algorithms". In: *Experimental Robotics III: 3rd International Symposium*. Ed. by T. Yoshikawa and F. Miyazaki. Vol. 200. Lecture Notes in Control and Information Sciences. London, United Kingdom: Springer, pp. 501–513. DOI: 10.1007/BFb0027617.

Mondada, F., Guignard, A., Bonani, M., Bar, D., Lauria, M., and Floreano, D. (2003). "SWARM-BOT: from concept to implementation". In: *2003 IEEE/RSJ International Conference On Intelligent Robots And Systems (IROS 2003)*. Vol. 2. Piscataway, NJ, USA: IEEE, pp. 1626–1631. DOI: 10.1109/IROS.2003.1248877.

Moscato, P. and Cotta, C. (2010). "A modern introduction to memetic algorithms". In: *Handbook of metaheuristics*. Ed. by M. Gendreau and J.-Y. Potvin. Vol. 146. International Series in Operations Research & Management Science. Boston,

MA, USA: Springer, pp. 141–183. DOI: 10.1007/978-1-4419-1665-5_6.

Muratore, F., Ramos, F., Turk, G., Yu, W., Gienger, M., and Peters, J. (2022). "Robot learning from randomized simulations: a review". In: *Frontiers in Robotics and AI* 9, p. 799893. DOI: 10.3389/frobt.2022.799893.

Neupane, A. and Goodrich, M. (2019). "Learning swarm behaviors using grammatical evolution and behavior trees". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. Ed. by S. Kraus. CA, USA: IJCAI Organization, pp. 513–520. DOI: 10.24963/ijcai.2019/73.

Nikolaev, A. G. and Jacobson, S. H. (2010). "Simulated annealing". In: *Handbook of metaheuristics*. Ed. by M. Gendreau and J.-Y. Potvin. Vol. 146. International Series in Operations Research & Management Science. Boston, MA, USA: Springer, pp. 1–39. DOI: 10.1007/978-1-4419-1665-5_1.

Nolfi, S. (2021). *Behavioral and Cognitive Robotics: An Adaptive Perspective*. Rome, Italy: Institute of Cognitive Sciences and Technologies, National Research Council.

Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. First. A Bradford Book. Cambridge, MA, USA: MIT Press.

Nouyan, S., Groß, R., Bonani, M., Mondada, F., and Dorigo, M. (2009). "Teamwork in self-organized robot colonies". In: *IEEE Transactions on Evolutionary Computation* 13.4, pp. 695–711. DOI: 10.1109/TEVC.2008.2011746.

O'Neill, M. and Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. First. Genetic Programming Series. Boston, MA, USA: Springer. DOI: 10.1007/978-1-4615-0447-4.

Ögren, P. (2012). "Increasing modularity of UAV control systems using computer game behavior trees". In: *AIAA Guidance, Navigation, and Control Conference*. Ed. by J. Thienel et al. Reston, VA, USA: AIAA Meeting Papers, pp. 358–393. DOI: 10.2514/6.2012-4458.

Oğuz, S., Heinrich, M. K., Allwright, M., Zhu, W., Wahby, M., Garone, E., and Dorigo, M. (2022). *S-drone: an open-source quadrotor for experimentation in swarm robotics*. Tech. rep. TR/IRIDIA/2022-010. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Osa, T., Pajarinen, J., Neumann, G., Bagnell, J. A., Abbeel, P., and Peters, J. (2018). "An algorithmic perspective on imitation learning". In: *Foundations and Trends® in Robot.* 7.1–2, pp. 1–179. DOI: 10.1561/2300000053.

Oung, R. and D'Andrea, R. (2011). "The distributed flight array". In: *Mechatronics* 21.6, pp. 908–917. DOI: 10.1016/j.mechatronics.2010.08.003.

Özdemir, A., Gauci, M., Bonnet, S., and Groß, R. (2018). "Finding consensus without computation". In: *IEEE Robotics and Automation Letters* 3.3, pp. 1346–1353. DOI: 10.1109/LRA.2018.2795640.

Özdemir, A., Gauci, M., and Groß, R. (2017). "Shepherding with robots that do not compute". In: *ECAL 2017, the Fourteenth European Conference on Artificial Life*. Cambridge, MA, USA: MIT Press, pp. 332–339. DOI: 10.7551/ecal_a_056.

Özdemir, A., Gauci, M., Kolling, A., Hall, M. D., and Groß, R. (2019). "Spatial coverage without computation". In: *2019 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 9674–9680. DOI: 10.1109/ICRA.2019.8793731.

Padberg, M. and Rinaldi, G. (1991). "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems". In: *SIAM Review* 33.1, pp. 60–100. DOI: 10.1137/1033004.

Paxton, C., Hundt, A., Jonathan, F., Guerin, K., and Hager, G. D. (2017). "CoSTAR: instructing collaborative robots with behavior trees and vision". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 564–571. DOI: 10.1109/ICRA.2017.7989070.

Payton, D., Daily, M., Estowski, R., Howard, M., and Lee, C. (2001). "Pheromone robotics". In: *Autonomous Robots* 11, pp. 319–324. DOI: 10.1023/A:1012411712038.

Perez, D., Nicolau, M., O'Neill, M., and Brabazon, A. (2011). "Evolving behaviour trees for the Mario AI competition using grammatical evolution". In: *Applications of Evolutionary Computation. EvoApplications 2011: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC*. Ed. by C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcázar, J. J. Merelo, F. Neri, M. Preuss, H. Richter, J. Togelius, and G. N. Yannakakis. Vol. 6624. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 123–132. DOI: 10.1007/978-3-642-20525-5_13.

Pérez Cáceres, L. (2017). "Automatic Algorithm Configuration: Analysis, Improvements and Applications". PhD thesis. Brussels, Belgium: Université Libre de Bruxelles.

Pérez Cáceres, L., Pagnozzi, F., Franzin, A., and Stützle, T. (2018). "Automatic configuration of GCC using irace". In: *Artificial Evolution. EA 2017*. Ed. by E. Lutton, P. Legrand, P. Parrend, N. Monmarché, and M. Schoenauer. Vol. 10764. LNCS. Cham, Switzerland: Springer, pp. 202–216. DOI: 10.1007/978-3-319-78133-4_15.

Pierce, D. (2018). *Robot Vacuums Are Finally Good—-Here's Which One to Buy*. https://www.wsj.com/articles/robot-vacuums-are-finally-good-heres-which-one-to-buy-11544968981. accessed on 2023-01-22.

Pinciroli, C. and Beltrame, G. (2016). "Buzz: a programming language for robot swarms". In: *IEEE Software* 33.4, pp. 97–100. DOI: 10.1109/MS.2016.95.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G. A., Ducatelle, F., Birattari, M., Gambardella, L. M., and Dorigo, M. (2012). "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems". In: *Swarm Intelligence* 6.4, pp. 271–295. DOI: 10.1007/s11721-012-0072-5.

Pini, G., Brutschy, A., Frison, M., Roli, A., Dorigo, M., and Birattari, M. (2011). "Task partitioning in swarms of robots: an adaptive method for strategy selection". In: *Swarm Intelligence* 5.3–4, pp. 283–304. DOI: 10.1007/s11721-011-0060-1.

Preiss, J. A., Honig, W., Sukhatme, G. S., and Ayanian, N. (2017). "Crazyswarm: a large nano-quadcopter swarm". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ, USA: IEEE, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376.

Prieto, A., Becerra, J. A., Bellas, F., and Duro, R. J. (2010). "Open-ended evolution as a means to self-organize heterogeneous multi-robot systems in real time". In: *Robotics and Autonomous Systems* 58.12, pp. 1282–1291. DOI: 10.1016/j.robot.2010.08.004.

Prorok, A., Correll, N., and Martinoli, A. (2011). "Multi-level spatial modeling for stochastic distributed robotic systems". In: *The International Journal of Robotics Research* 30.5, pp. 574–589. DOI: 10.1177/0278364911399521.

Prorok, A., Hsieh, M. A., and Kumar, V. (2017). "The impact of diversity on optimal control policies for heterogeneous robot swarms". In: *IEEE Transactions on Robotics* 33.2, pp. 346–358. DOI: 10.1109/TRO.2016.2631593.

Quinn, M., Smith, L., Mayley, G., and Husbands, P. (2003). "Evolving controllers for a homogeneous system of physical robots: structured cooperation with minimal sensors". In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 361.1811, pp. 2321–2343. DOI: 10.1098/rsta.2003.1258.

R Development Core Team (2008). *The R Project for Statistical Computing*. http://www.R-project.org.

Ramos, R. P., Oliveira, S. M., Vieira, S. M., and Christensen, A. L. (2019). "Evolving flocking in embodied agents based on local and global application of Reynolds' rules". In: *PLOS ONE* 14.10, e0224376. DOI: 10.1371/journal.pone.0224376.

Reina, A., Bose, T., Trianni, V., and Marshall, J. A. R. (2018). "Effects of spatiality on value-sensitive decisions made by robot swarms". In: *Distributed Autonomous Robotic Systems: The 13th International Symposium*. Ed. by R. Groß, A. Kolling, S. Berman, E. Frazzoli, A. Martinoli, F. Matsuno, and M.

Gauci. Vol. 6. Springer Proceedings in Advanced Robotics. Springer, pp. 461–473. DOI: 10.1007/978-3-319-73008-0_32.

Reina, A., Valentini, G., Fernández-Oto, C., Dorigo, M., and Trianni, V. (2015). "A design pattern for decentralised decision making". In: *PLOS ONE* 10.10, e0140950. DOI: 10.1371/journal.pone.0140950.

Rubenstein, M., Cornejo, A., and Nagpal, R. (2014). "Programmable self-assembly in a thousand-robot swarm". In: *Science* 345.6198, pp. 795–799. DOI: 10.1126/science.1254295.

Şahin, E. (2005). "Swarm robotics: from sources of inspiration to domains of application". In: *Swarm Robotics: SAB 2004 International Workshop*. Ed. by E. Şahin and W. M. Spears. Vol. 3342. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 10–20. DOI: 10.1007/978-3-540-30552-1_2.

Salman, M., Ligot, A., and Birattari, M. (2019). "Concurrent design of control software and configuration of hardware for robot swarms under economic constraints". In: *PeerJ Computer Science* 5, e221. DOI: 10.7717/peerj-cs.221.

Schede, E., Brandt, J., Tornede, A., Wever, M., Bengs, V., Hüllermeier, E., and Tierney, K. (2022). "A survey of methods for automated algorithm configuration". In: *Journal of Artificial Intelligence Research* 75, pp. 425–487. DOI: 10.1613/jair.1.13676.

Schmickl, T. and Hamann, H. (2011). "BEECLUST: a swarm algorithm derived from honeybees". In: *Bio-inspired Computing and Networking*. Ed. by Y. Xiao. Boca Raton, FL, USA: CRC Press, pp. 95–137. DOI: 10.1201/b10781.

Schmickl, T., Thenius, R., Moslinger, C., Timmis, J., Tyrrell, A., Read, M., Hilder, J., Halloy, J., Campo, A., Stefanini, C., Manfredi, L., Orofino, S., Kernbach, S., Dipper, T., and Sutantyo, D. (2011). "CoCoRo – the self-aware underwater swarm". In: *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. Ed. by A. Lomuscio, P. Scerri, A. Bazzan, and M. Huhns. Piscataway, NJ, USA: IEEE, pp. 120–126. DOI: 10.1109/SASOW.2011.11.

Schranz, M., Di Caro, G. A., Schmickl, T., Elmenreich, W., Arvin, F., Şekercioğlu, Y. A., and Sende, M. (2021). "Swarm intelligence and cyber-physical systems: concepts, challenges and future trends". In: *Swarm and Evolutionary Computation* 60.14, p. 100762. DOI: 10.1016/j.swevo.2020.100762.

Schranz, M., Umlauft, M., Sende, M., and Elmenreich, W. (2020). "Swarm robotic behaviors and current applications". In: *Frontiers in Robotics and AI* 7, p. 36. DOI: 10.3389/frobt.2020.00036.

Silva, F., Correia, L., and Christensen, A. L. (2017). "Evolutionary online behaviour learning and adaptation in real robots". In: *Royal Society Open Science* 4.7, p. 160938. DOI: 10.1098/rsos.160938.

Silva, F., Duarte, M., Correia, L., Oliveira, S. M., and Christensen, A. L. (2016). "Open issues in evolutionary robotics". In: *Evolutionary Computation* 24.2, pp. 205–236. DOI: 10.1162/EVCO_a_00172.

Silva, F., Urbano, P., Correia, L., and Christensen, A. L. (2015). "odNEAT: an algorithm for decentralised online evolution of robotic controllers". In: *Evolutionary Computation* 23.3, pp. 421–449. DOI: 10.1162/EVCO_a_00141.

Slavkov, I., Carrillo-Zapata, D., Carranza, N., Diego, X., Jansson, F., Kaandorp, J., Hauert, S., and Sharpe, J. (2018). "Morphogenesis in robot swarms". In: *Science Robotics* 3.25, eaau9178. DOI: 10.1126/scirobotics.aau9178.

Soares, J. M., Navarro, I., and Martinoli, A. (2015). "The Khepera IV mobile robot: performance evaluation, sensory data and software toolbox". In: *Robot 2015: Second Iberian Robotics Conference: Advances in Robotics, Volume 1*. Ed. by L. P. Reis, A. P. Moreira, P. U. Lima, L. Montana, and V. F. Muñoz Martínez. Vol. 417. Advances in Intelligent Systems and Computing. Cham, Switzerland: Springer, pp. 767–781. DOI: 10.1007/978-3-319-27146-0_59.

Šošić, A., Khuda Bukhsh, W. R., Zoubir, A. M., and Koeppl, H. (2017). "Inverse reinforcement learning in swarm systems". In: *AAMAS '17: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1413–1421.

Soysal, O. and Şahin, E. (2005). "Probabilistic aggregation strategies in swarm robotic systems". In: *2005 IEEE Swarm Intelligence Symposium, SIS 2005*. Piscataway, NJ, USA: IEEE, pp. 325–332. DOI: 10.1109/SIS.2005.1501639.

Soysal, O. and Şahin, E. (2007). "A macroscopic model for self-organized aggregation in swarm robotic systems". In: *Swarm Robotics: Second International Workshop, SAB 2006*. Ed. by E. Şahin, W. M. Spears, and A. F. T. Winfield. Vol. 4433. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 27–42. DOI: 10.1007/978-3-540-71541-2_3.

Spaey, G., Kegeleirs, M., Garzón Ramos, D., and Birattari, M. (2020). "Evaluation of alternative exploration schemes in the automatic modular design of robot swarms". In: *Artificial Intelligence and Machine Learning: BNAIC 2019, BENELEARN 2019*. Ed. by B. Bogaerts, G. Bontempi, P. Geurts, N. Harley, B. Lebichot, T. Lenaerts, and G. Louppe. Vol. 1196. Communications in Computer and Information Science. Cham, Switzerland: Springer, pp. 18–33. DOI: 10.1007/978-3-030-65154-1_2.

Spears, W. M. and Spears, D., eds. (2012). *Physicomimetics: Physics-Based Swarm Intelligence*. Berlin, Germany: Springer. DOI: 10.1007/978-3-642-22804-9.

Spears, W. M., Spears, D., Heil, R., Kerr, W., and Hettiarachchi, S. (2005). "An overview of physicomimetics". In: *Swarm Robotics: SAB 2004 International*

Workshop. Ed. by E. Şahin and W. M. Spears. Vol. 3342. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 84–97.

Stanley, K. O. and Miikkulainen, R. (2002). "Evolving neural networks through augmenting topologies". In: *Evolutionary Computation* 10.2, pp. 99–127. DOI: 10.1162/106365602320169811.

Stranieri, A., Turgut, A. E., Salvaro, M., Garattoni, L., Francesca, G., Reina, A., Dorigo, M., and Birattari, M. (2013). *IRIDIA's arena tracking system*. Tech. rep. TR/IRIDIA/2013-013. Brussels, Belgium: IRIDIA, Université Libre de Bruxelles.

Sugawara, K. and Sano, M. (1997). "Cooperative acceleration of task performance: foraging behavior of interacting multi-robots system". In: *Physica D: Nonlinear Phenomena* 100.3-4, pp. 343–354. DOI: 10.1016/S0167-2789(96)00195-9.

Talamali, M. S., Saha, A., Marshall, J. A. R., and Reina, A. (2021). "When less is more: robot swarms adapt better to changes with constrained communication". In: *Science Robotics* 6.56, eabf1416. DOI: 10.1126/scirobotics.abf1416.

The Creative Cloud Team (Adobe) (2017). *Extending Your Reality: What Happens When You Let a Drone Carry Your Camera Away*. https://blog.adobe.com/en/2017/12/04/extending-reality-happens-let-drone-carry-camera-away. accessed on 2023-01-07.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton Brown, K. (2013). "Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms". In: *KDD '13: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. Ed. by R. Ghani, T. E. S. abd Paul Bradley, R. Parekh, and J. He. New York, NY, USA: ACM, pp. 847–855. DOI: 10.1145/2487575.2487629.

Trianni, V. (2008). *Evolutionary Swarm Robotics*. Berlin, Germany: Springer. DOI: 10.1007/978-3-540-77612-3.

Trianni, V., Groß, R., Labella Thomas, H., Şahin, E., and Dorigo, M. (2003). "Evolving aggregation behaviors in a swarm of robots". In: *Advances in Artificial Life: 7th European Conference, ECAL 2003*. Ed. by W. Banzhaf, J. Ziegler, T. Christaller, P. Dittrich, and J. T. Kim. Vol. 2801. Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 865–874. DOI: 10.1007/978-3-540-39432-7_93.

Trianni, V. and López-Ibáñez, M. (2015). "Advantages of task-specific multi-objective optimisation in evolutionary robotics". In: *PLOS ONE* 10.8, e0136406. DOI: 10.1371/journal.pone.0136406.

Trianni, V. and Nolfi, S. (2011). "Engineering the evolution of self-organizing behaviors in swarm robotics: a case study". In: *Artificial Life* 17.3, pp. 183–202. DOI: 10.1162/artl_a_00031.

Trianni, V., Tuci, E., Ampatzis, C., and Dorigo, M. (2014). "Evolutionary swarm robotics: a theoretical and methodological itinerary from individual neuro-controllers to collective behaviours". In: *The Horizons of Evolutionary Robotics*. Ed. by P. A. Vargas, E. A. Di Paolo, I. Harvey, and P. Husbands. Boston, MA, USA: MIT Press, pp. 153–178. DOI: 10.7551/mitpress/8493.003.0008.

Ugur, E., Turgut, A. E., and Şahin, E. (2007). "Dispersion of a swarm of robots based on realistic wireless intensity signals". In: *2007 22nd International Symposium on Computer and Information Sciences (ISCIS)*. Ed. by E. G. Schmidt, İ. Ulusoy, N. K. Çiçekli, U. Halıcı, and M. V. Atalay. Piscataway, NJ, USA: IEEE, pp. 1–6. DOI: 10.1109/ISCIS.2007.4456899.

Valentini, G., Ferrante, E., Hamann, H., and Dorigo, M. (2016). "Collective decision with 100 Kilobots: speed versus accuracy in binary discrimination problems". In: *Autonomous Agents and Multi-Agent Systems* 30.3, pp. 553–580. DOI: 10.1007/s10458-015-9323-3.

Valentini, G., Hamann, H., and Dorigo, M. (2014). "Self-organized collective decision making: the weighted voter model". In: *AAMAS '14: Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 45–52.

Vigelius, M., Meyer, B., and Pascoe, G. (2014). "Multiscale modelling and analysis of collective decision making in swarm robotics". In: *PLOS ONE* 9.11, e111542. DOI: 10.1371/journal.pone.0111542.

Vitanza, A., Rosseti, P., Mondada, F., and Trianni, V. (2019). "Robot swarms as an educational tool: the Thymio's way". In: *International Journal of Advanced Robotic Systems*, pp. 1–13. DOI: 10.1177/1729881418825186.

wallonia.be (2014). *An Oscar for Flying-Cam!* https://www.wallonia.be/en/news/un-oscar-pour-flying-cam. accessed on 2023-01-07.

Walter, W. G. (1950). "An imitation of life". In: *Scientific American* 182.5, pp. 42–45.

Watson, R. A., Ficici, S. G., and Pollack, J. B. (2002). "Embodied evolution: distributing an evolutionary algorithm in a population of robots". In: *Robotics and Autonomous Systems* 39.1, pp. 1–18. DOI: 10.1016/S0921-8890(02)00170-7.

Werfel, J., Petersen, K., and Nagpal, R. (2014). "Designing collective behavior in a termite-inspired robot construction team". In: *Science* 343.6172, pp. 754–758. DOI: 10.1126/science.1245842.

Winfield, A. F. T., Harper, C. J., and Nembrini, J. (2005a). "Towards dependable swarms and a new discipline of swarm engineering". In: *Swarm Robotics: SAB 2004 International Workshop*. Ed. by E. Şahin and W. M. Spears. Vol. 3342.

Lecture Notes in Computer Science. Berlin, Germany: Springer, pp. 126–142. DOI: 10.1007/978-3-540-30552-1_11.

Winfield, A. F. T., Sa, J., Fernández-Gago, M. C., Dixon, C., and Fisher, M. (2005b). "On formal specification of emergent behaviours in swarm robotic systems". In: *International Journal of Advanced Robotic Systems* 2.4. DOI: 10.5772/5769.

Witze, A. (2022). "NASA's Perseverance rover begins key search for life on Mars". In: *Nature* 606, pp. 441–442. DOI: 10.1038/d41586-022-01543-z.

Xie, H., Sun, M., Fan, X., Lin, Z., Chen, W., Wang, L., Dong, L., and He, Q. (2019). "Reconfigurable magnetic microrobot swarm: multimode transformation, locomotion, and manipulation". In: *Science Robotics* 4.28, eaav8006. DOI: 10.1126/scirobotics.aav8006.

Yamins, D. and Nagpal, R. (2008). "Automated global-to-local programming in 1-D spatial multi-agent systems". In: *AAMAS '08: The Seventh International Conference on Autonomous Agents and Multiagent Systems*. Ed. by L. Padgham, D. Parkes, J. Müller, and S. Parsons. Vol. 2. Richland, SC, USA: International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 615–622.

Yang, G.-Z., Bellingham, J., Dupont, P. E., Fischer, P., Floridi, L., Full, R., Jacobstein, N., Kumar, V., McNutt, M., Merrifield, R., Nelson, B. J., Scassellati, B., Taddeo, M., Taylor, R., Veloso, M., Wang, Z. L., and Wood, R. J. (2018). "The grand challenges of Science Robotics". In: *Science Robotics* 3.14, eaar7650. DOI: 10.1126/scirobotics.aar7650.

Yu, J., Wang, B., Du, X., Wang, Q., and Zhang, L. (2018). "Ultra-extensible ribbon-like magnetic microswarm". In: *Nature Communications* 9.1, p. 3260. DOI: 10.1038/s41467-018-05749-6.

Zhao, W., Queralta, J. P., and Westerlund, T. (2020). "Sim-to-real transfer in deep reinforcement learning for robotics: a survey". In: *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*. Piscataway, NJ, USA: IEEE, pp. 737–744. DOI: 10.1109/SSCI47803.2020.9308468.