This is a preprint. The final authenticated version is available online at:
https://doi.org/10.1007/978-3-030-00533-7_3

Behavior Trees as a Control Architecture in the Automatic Modular Design of Robot Swarms*

Jonas Kuckling[0000−0003−2391−2275], Antoine Ligot[0000−0001−7388−2866], Darko Bozhinoski[0000−0002−6853−0310], and Mauro Birattari[0000−0003−3309−2194]

IRIDIA, Université libre de Bruxelles, Belgium
mbiro@ulb.ac.be

Abstract. Previous research has shown that automatically combining low-level behaviors into a probabilistic finite state machine produces control software that crosses the reality gap satisfactorily. In this paper, we explore the possibility of adopting behavior trees as an architecture for the control software of robot swarms. We introduce `Maple`: an automatic design method that combines preexisting modules into behavior trees. To highlight the potential of this control architecture, we present robot experiments in which we compare `Maple` with `Chocolate` and `EvoStick` on two missions: FORAGING and AGGREGATION. `Chocolate` and `EvoStick` are two previously published automatic design methods. `Chocolate` is a modular method that generates probabilistic finite state machines and `EvoStick` is a traditional evolutionary robotics method. The results of the experiments indicate that behavior trees are a viable and promising architecture to automatically generate control software for robot swarms.

1 Introduction

In swarm robotics, a group of simple robots works together to achieve a common goal that is beyond the capabilities of a single robot [34, 2, 11, 5, 19, 4]. The collective behavior of the swarm is the result of the local interactions that each robot has with its neighboring peers and with the environment. One of the biggest challenges is to conceive the control software of the individual robots [11]. Often control software is designed manually in a trial-and-error process [5]. This approach is time-consuming, prone to error and bias and difficult to replicate [14, 4]. A promising alternative is automatic design. In automatic design, the design problem is transformed into an optimization problem. The design space of the possible instances of control software is mapped into a solution space on which an optimization algorithm searches a solution that maximizes a mission-dependent performance measure. Due to numerous constraints (of which time and hardware properties are the most notable ones), automatic design is often

* JK and AL contributed equally to the research and should be considered co–first authors. Behavior trees were originally brought to the attention of the authors by DB. The proposed method was conceived by the four authors. It was implemented and tested by JK and AL. The initial draft of the manuscript was written by JK and AL and then revised by DB and MB. The research was directed by MB.

performed in simulation. As simulation is unavoidably only an approximation of reality, the so-called reality gap has to be faced by control software developed in simulation. It has been observed that different design methods might be more or less robust to the reality gap [16]. When assessing an automatic design method, it is therefore fundamental to perform tests with real robots to study its ability to cross the reality gap satisfactorily.

A popular approach to the automatic design of robot control software is evolutionary robotics [13]. Evolutionary swarm robotics is the application of evolutionary algorithms to generate control software for swarm robotics [37]. In this approach, robots are controlled by artificial neural networks that map sensor readings to commands that are fed to the actuators. Other approaches have been proposed that generate control software by assembling predefined modules. For example, Duarte et al. [12] generated a set of neural networks to perform low-level actions. These neural networks were then combined into a finite state machine. The benefit of the approach is that it is easier generate multiple neural networks that perform low-level actions rather than a single one that performs the whole mission. The limitation is that the designer still needs to decompose the task into suitable subtasks. Francesca et al. [16, 15] defined AutoMoDe, a method in which a set of preexisting mission-agnostic *constituent behaviors* and *conditions* are assembled into a finite state-machine by an optimization process that maximizes a mission-specific performance measure. The authors developed two instances of AutoMoDe: `Vanilla` [16] and `Chocolate` [15], which differ in the optimization algorithm adopted. They compared them with a standard evolutionary method they called `EvoStick` [16]. While `EvoStick` performs better in simulation, `Vanilla` and `Chocolate` proved to be more robust to the reality gap and obtain better results in reality.

In this paper, we explore the possibility of automatically assembling preexisting modules into a behavior tree. Behavior trees are a control architecture that was initially developed as an alternative to finite state machines for specifying the behavior of non-player characters in video games [23, 8]. Behavior trees gained popularity in the video game industry mainly because of their inherent modularity [8]. Subsequently, they attracted the attention of the academia, mostly in the robotics domain [9]. Compared to finite state machines, behavior trees promote increased readability, maintainabilty and code reuse [10].

Behavior trees are a promising control architecture to be adopted in swarm robotics. Indeed, they can be seen as generalizations of three classical architecture already studied in the literature: the subsumption architecture [6], sequential behavior compositions [7], and decision trees [30].

In this paper, we show that behavior trees can be used as a control architecture in the automatic design of robot swarms. We propose `Maple`, an automatic design method that fine-tunes and assembles preexisting modules (constituent behaviors and conditions) into a behavior tree. We present the results of experiments in which we automatically design control software for two missions: FORAGING and AGGREGATION. In our experiments, `Maple` outperforms `EvoStick` and obtains results that are comparable with those of `Chocolate`.

The research presented in this article prompts us to reconsider the original definition of AutoMoDe, which arbitrarily restricts AutoMoDe to the generation of probabilistic finite state machines [16]. The defining purpose of AutoMoDe–*automatic modular design* – is to generate control software for robot swarms by assembling and fine-tuning preexisting modules. The architecture into which modules are assembled is a secondary issue which we find it should not limit the methods defined as automatic modular design. In the following, we will consider as instances of AutoMoDe all methods that assemble and fine-tune preexisting modules, irrespective of the architecture into which they are cast and of the optimization algorithm used to generate the solutions. In this precise sense, we consider `Maple` to be an instance of AutoMoDe.

The paper is structured as follows: Section 2 provides an overview of the behavior tree architecture. Section 3 introduces `Maple`. Section 4 describe the experimental setup and Section 5 presents the results. Section 6 discussed related research and Section 7 concludes the paper and sketches future developments.

2 Behavior Trees

Behavior trees have been used as an alternative to finite state machines [27]. In this paper, we follow the definition given by Marzinotto et al. [27]. A behavior tree is a tree structure that contains one root node, control nodes, and execution nodes (actions or conditions). Execution is controlled by a tick generated by the root and propagated through the tree. When ticked by its parent, a node is activated. After execution, it returns one of three possible values: *success*, *running*, or *failure*. Condition nodes that are ticked observe the world state and return *success*, if their condition is fulfilled; and *failure*, otherwise. Action nodes that are ticked returns *success*, if their action is completed; *failure*, if their action cannot be completed; and *running*, if their action is still in progress. Control nodes distribute the tick to their children. Their return value depends on those returned by the children. There are six different types of control nodes: selector, selector*, sequence, sequence*, parallel and decorator—see Table 1.

Additionally, behavior trees implement the principle of two-way control transfers [31]. Not only can control be passed from a parent node to its child node, but the child can return execution to its parent, along with information about the state of execution. In a finite state machine, the control flow is only one-directional, that is, a state cannot return control to the predecessor.

Perhaps the most important property of behavior trees is their inherent modularity [10]. Each subtree of a behavior tree is, by definition, a valid behavior tree as well. Thanks to the modularity, it is possible to adjust, remove, or add subtrees without having to account for new or missing interactions [31]. Combining subtree modules in a behavior tree leads to a hierarchical structure, which can simplify the analysis, for both humans and computers [10].

The aforementioned properties are appealing in the automatic design of control software for robot swarms. The enhanced expressiveness and the two-way control transfers could allow the representation of behaviors that cannot be eas-

Table 1. Overview of possible control nodes in a behavior tree.

Name	Symbol	Description
selector	?	Ticks children sequentially as long as they return *failure*.
selector*	?*	Ticks children sequentially as long as they return *failure*. Resumes ticking at last ticked node, if it returned *running*.
sequence	$\rightarrow$	Ticks children sequentially as long as they return *success*.
sequence*	$\rightarrow^*$	Ticks children sequentially as long as they return *success*. Resumes ticking at last ticked node, if it returned *running*.
parallel	$\rightrightarrows$	Ticks all children simultaneously. Returns *success* (or *failure*), if a majority of the children return *success* (or *failure*). Otherwise it returns *running*.
decorator	δ	Executes a custom function on its only child. The function can either manipulate the number of ticks given to the child, or the value returned to the parent.

ily implemented using finite state machines. The structural modularity could greatly simplify the implementation of optimization algorithms based on local manipulations. It could also allow pruning unused parts to increment readability. Finally, subtrees could be optimized independently of each other and used afterwards as building blocks to generate more complex behaviors.

3 AutoMoDe-`Maple`

`Maple` is an automatic design method that, by combining and fine-tuning preexisting modules, generates control software in the form of behavior trees. In defining `Maple`, our goal was to explore the possibility of using behavior trees in the modular design of control software for robot swarms. We wished to define a method that we could then compare with `Chocolate`, the existing state-of-the-art in modular design, which generates finite state machines. We thought that, at this stage of our research, the comparison would have been the most informative if we reduced the differences between `Maple` and `Chocolate` as much as possible, so as to isolate the element we wished to study: the architecture. Therefore, we conceived `Maple` so that it shares with `Chocolate` the modules to be assembled and the optimization algorithm. The only difference between `Maple` and `Chocolate` is the architecture: behavior trees for the former, finite state machines for the latter.

The modules assembled by `Maple` are those used by both `Vanilla` [16] and `Chocolate` [15]. To use these modules within a behavior tree, we included in `Maple` only a subset of the control nodes described in Section 2.

3.1 Robotic Platform

`Maple` generates control software for an extended version of e-puck [29, 18]. Formally, the subset of sensors and actuators that are used by `Maple`, along with the

Table 2. Reference model RM 1.1 [21]. Sensors and actuators of the extended version of the e-puck robot. Period of control cycle: 100 ms.

sensor/actuator	variables	values
proximity	$prox_i$, with $i \in \{0, ..., 7\}$	$[0, 1]$
light	$light_i$, with $i \in \{0, ..., 7\}$	$[0, 1]$
ground	$ground_i$, with $i \in \{0, ..., 2\}$	$\{black, gray, white\}$
range-and-bearing	n	$\{0, ..., 19\}$
	V_d	$([0, 0.7]\,\text{m}, [0, 2\pi]\,\text{radian})$
wheels	v_l, v_r	$[-0.12, 0.12]\,\text{m/s}$

corresponding variables, are defined by the reference model RM 1.1 [21], which we reproduce in Table 2 for the convenience of the reader.

The e-puck is a two wheeled robot. The control software can adjust the velocity of the motors of each wheel (v_r and v_l). The e-puck can detect the presence of nearby obstacles ($prox_i$), measure ambient light ($light_i$), and tell whether the floor situated directly beneath itself is white, gray, or black ($ground_i$). Finally, thanks to its range-and-bearing board [20] the e-puck is aware of the presence of its peers in a range of up to 0.7 m: it knows their number (n) and a vector V_d indicating the direction of attraction to the neighboring peers, following the framework of virtual physics [35].

3.2 Set of Modules

`Maple` uses the set of preexisting modules originally defined for `Vanilla` [16]. The set is composed of six low-level behaviors (i.e., activities performed by the robot) and six conditions (i.e., assessments of particular situations experienced by the robot). In a behavior tree, a leaf node is either an action or a condition. `Maple` selects the action nodes among the set of `Vanilla`'s low-level behaviors, and the condition nodes among the set of `Vanilla`'s conditions. In this section, we briefly describe `Vanilla`'s low-level behaviors and conditions. We refer the reader to the work of Francesca et al. [16] for more details.

Low-level Behaviors. *Exploration* is a random walk strategy. The robot goes straight until an obstacle is perceived by the front proximity sensors. Then, the robot turns on the spot for a random number of control cycles drawn in $\{0, ..., \tau\}$, where τ is an integer parameter $\in \{0, ..., 100\}$. *Stop* orders the robot to stay still. *Phototaxis* moves the robot towards a light source. If no light source is perceived, the robot goes straight. *Anti-phototaxis* moves the robot away from the light source [1]. If no light source is perceived, the robot goes straight. *Attraction* moves the robot in the direction of the neighboring peers (V_d). The speed of convergence towards the detected peers is controlled by a real parameter $\alpha \in [1, 5]$. If no peer is detected, the robot goes straight. *Repulsion* moves the robot away from the

[1] In biology this behavior is known as *negative phototaxis* [28].

neighboring peers ($-V_d$). The real parameter $\alpha \in [1, 5]$ controls the speed of divergence. Obstacle avoidance is embedded in all low-level behaviors, with the exception of stop. As stated earlier, this is not a design choice we made for `Maple` but rather an earlier decision from `Vanilla` [16] that is kept to allow comparison with previosly obtained results. The parameters τ and α must be tuned by the automatic design process.

Conditions. *Black-*, *gray-* and *white-floor* are true with probability $\beta \in [0, 1]$ if the ground sensor perceives the floor as black, gray, or white, respectively. *Neighbor-count* is true with a probability computed as a function $z(n) \in [0, 1]$ of the number of robots detected via the range-and-bearing board. A real parameter $\eta \in [0, 20]$ and an integer parameter $\xi \in \{0, ..., 10\}$ control the steepness and the inflection point of the function, respectively. *Inverted-neighbor-count* is true with probability $1 - z(n)$. *Fixed-probability* is true with probability $\beta \in [0, 1]$. The parameters β, η and ξ must be tuned by the automatic design process.

3.3 Control Software Architecture

We use the preexisting low-level behaviors of `Vanilla` [16] without any modification. In the traditional implementation of behavior trees, an action node is able to tell whether the system it controls (i) successfully executed, (ii) is still executing, or (iii) failed to execute the required activity. The action node then returns the corresponding state variable (i.e., *success*, *running*, or *failure*).

The low-level behaviors of `Vanilla` were designed to be used as states of probabilistic finite state machines, and were meant to be executed until an external condition was enabled. Because of their implementations, when used as action nodes within `Maple`, the low-level behaviors can only return *running*. As a consequence, part of the control-flow nodes of behavior trees do not work as intended. For example, a sequence node with two `Vanilla`'s behaviors as children would always directly return *running* after the first behavior is executed once, and would never execute the second one—see Table 1.

To use `Vanilla`'s behaviors as action nodes, `Maple` instantiates behavior trees that have a restricted topology and use only a subset of all available control nodes. The root node must be of the type sequence* and can only have selector nodes as children. Within `Maple`, each subtree defined by a selector node is forced to have two children: a condition node as the left child, and an action node as the right child. In order to stay close to `Vanilla`'s restriction of a maximum of four states in the finite state machine, the behavior tree is allowed to contain a maximum of four selector subtrees. Figure 1 illustrates an example (with only three out of four possible subtrees) of the restricted topology of the behavior trees that `Maple` can produce. In this example, action node A_1 is executed as long as condition node C_1 returns *failure*. When condition node C_1 returns *success*, the sequence* node ticks the next selector subtree, and so forth. Similarly to `Chocolate` [15], `Maple` uses Iterated F-race [26] as the optimization algorithm to search for the best possible instance of behavior tree among all the possible ones.

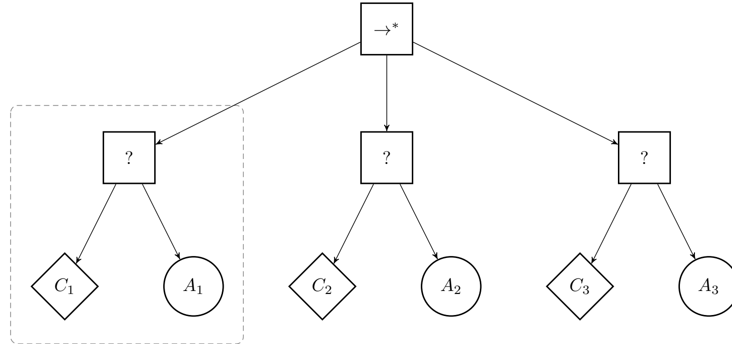

Fig. 1. Illustration of a behavior tree that can be generated by `Maple`. `Maple` determines the number of selector subtrees (highlighted by the dashed box) and specifies the condition and action nodes for each of them. The type of the root node is predefined.

4 Experimental Setup

In this section, we describe the automatic design methods under analysis, the missions on which we test them, and the protocol we follow.

4.1 Automatic design methods

We compare `Maple` with `Chocolate` [15] and `EvoStick` [16, 15]. As `Maple`, `Chocolate` and `EvoStick` are based on reference model RM1.1. We briefly describe these methods and we refer the reader to Francesca et al. [16, 15] for the details.

`Chocolate` selects, fine-tunes, and combines preexisting modules into probabilistic finite state machines. It uses the same twelve modules as `Vanilla` and `Maple`. `Chocolate` is restricted to create probabilistic finite state machines comprising up to four states and up to four outgoing edges per state. Similarly to `Maple`, `Chocolate` uses Iterated F-race [26] as optimization algorithm.

`EvoStick` is an implementation of the evolutionary robotics approach: the topology of a neural network is fixed, and an evolutionary algorithm is used to optimize the weights of the connections. The network considered in `EvoStick` is fully connected, feed-forward and does not contain hidden neurons. It comprises 24 input nodes for the readings of the sensors described in the reference model RM 1.1: 8 for the proximity sensors, 8 for the light sensors, 3 for the ground sensors, and 5 for the range-and-bearing board. Out of the 5 input nodes dedicated to the range-and-bearing board, one is allocated to the number of neighbors, and the four others to the scalar projections of the vector pointing to the center of mass of these neighbors on four unit vectors. The neural network comprises 2 output nodes for the velocities of the left and right wheels.

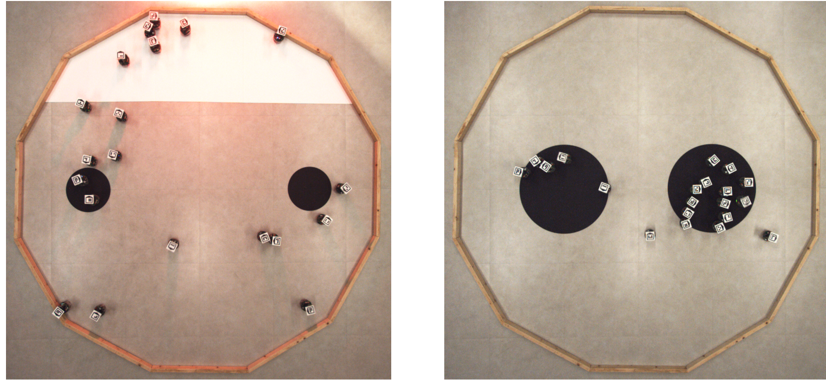

Fig. 2. FORAGING (*left*) and AGGREGATION (*right*).

4.2 Missions

The missions considered are FORAGING and AGGREGATION. They have already been studied in [16]. We refer the reader to the original article for the details. In the two missions, the robots operate in a dodecagonal arena delimited by walls and covering an area of $4.91\,\mathrm{m}^2$. We limit the duration of the missions to $120\,\mathrm{s}$.

FORAGING. The arena contains two source areas (black circles) and a nest (white area). A light is placed behind the nest to help the robots to navigate (Figure 2, left). In this idealized version of foraging, a robot is deemed to retrieve an object when it enters a source and then the nest. The goal of the swarm is to retrieve as many objects as possible. The objective function is $F_{\mathrm{F}} = N_i$, where N_i is the number of objects retrieved.

AGGREGATION. The swarm must select one of the two black areas and aggregate there [17, 16] (Figure 2, right). The objective function is $F_{\mathrm{A}} = \max(N_l, N_r)/N$, where N_l and N_r are the number of robots located on the left and right area, respectively; and N is the total number of robots. The objective function is computed at the end of the run, and is maximized when all robots are either on the left or the right area.

4.3 Protocol

We considered a robot swarm composed of 20 e-pucks. The three automatic design methods—`Maple`, `Chocolate` and `EvoStick`—produce control software for two missions—FORAGING and AGGREGATION. Since all the design methods are stochastic, for each mission, each design method is executed 10 times and produces 10 instances of control software. The design budget allocated to each method for each mission is 50 000 simulation runs: this is the maximum number of simulation runs allowed during the design process. To study the generalization

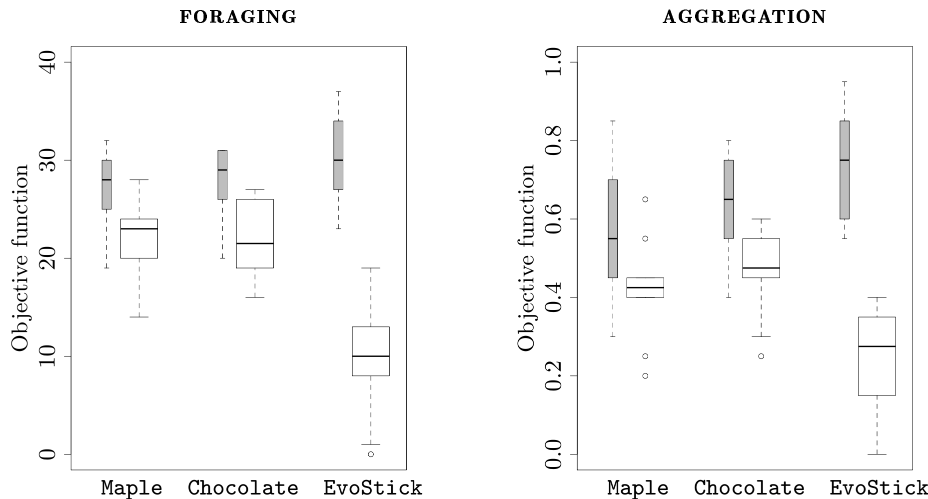

Fig. 3. Results of the experiments. The gray boxes represent the performance assessed in simulation; the white boxes represent the performance assessed in reality.

capabilities of the design methods, we assess the performance of each instance of control software once in simulation, and once in reality [3]. All simulations are performed using ARGoS3, beta 48. [33, 18].

In reality, to automatically measure the performance of the swarm, we use a system composed of an overhead camera and markers on the robots to track their position in real time [36]. Experimental runs start from 10 different initial positions/orientations of the robots. We use the tracking system to automatically guide the robots to the initial position/orientation of each run. During a run, we interfere with the robots only if they tip over due to a collision. In this case, we intervene and put them upright to avoid damages.

We present the results in the form of box-and-whiskers boxplots. For each method and each mission, we report two boxplots: one for simulation and one for reality. In the following, statements like "method A is significantly better than B," always imply we performed a Wilcoxon rank-sum test that detected significance with confidence of at least 95%.

5 Results

In this section, we report the results for each mission considered. The instances of control software produced, the details of their performances both in simulation and in reality, and videos of their execution on the robots are available as online supplementary material [25].

FORAGING. Although the performance of control software produced by the three automatic design methods is similar in simulation, `Maple` and `Chocolate` are significantly better than `EvoStick` in reality. The performance of all three methods

drops significantly when passing from simulation to reality, but `EvoStick` suffers from the reality gap the most. See Figure 3(*left*).

`Maple` and `Chocolate` generate control software that displays expected and similar strategies: the robots explore the environment and once a source of food (i.e., a black area) is found, they navigate towards the nest (i.e., the white area) guided by the light. The performance drop that affects `Maple` and `Chocolate` when porting the control software from simulation to reality is probably due to the fact that simulation does not properly reproduce the frictions experienced by the robots. In reality, due to friction, robots become sometimes unable to move and therefore do not contribute to the foraging process. Contrarily to `Maple` and `Chocolate`, and with the exception of a few cases, `EvoStick` was unable to generate instances of control software that display an effective foraging behavior in reality. Indeed, in most cases, the robots seem unable to navigate efficiently.

AGGREGATION. In simulation, `Maple` and `Chocolate` show similar performance, but `EvoStick` performs significantly better than `Maple`. Also in reality, `Maple` and `Chocolate` perform similarly, but they are both significantly better than `EvoStick`. Indeed, the performance of `EvoStick` drops considerably from simulation to reality, whereas the performance drop of `Maple` and `Chocolate` is smaller. See Figure 3(*right*).

The instances of control software produced by `Maple` and `Chocolate` are able to find the black areas and stop there. Contrarily, the instances of control software produced by `EvoStick` do not efficiently search the space. When a black area is found, the robots tend to leave it quickly. Neither of the three methods produced control software that displayed effective collective decision making.

6 Related Work

Most of the early research on behavior trees has concentrated on their use for game development [32, 1]. Subsequently, research has been devoted to the application of behavior trees in robotics. For example, Marzinotto et al. [27] manually designed a behavior tree for manipulation on the NAO robot. Hu et al. [22] described an application of behavior trees to semi-autonomous, simulated surgery.

Jones et al. [24] proposed an automatic design method for robot swarms in which the control architecture of robots is a behavior tree. To the best of our knowledge, that is the first and only application of behavior trees in swarm robotics. The authors used genetic programming to generate control software for kilobots in a foraging mission. The action nodes are atomic commands, such as setting motor state, storing information, or broadcasting a signal. The results show that behavior trees can be effectively used to control the robots of a swarm and that the control software generated is human-readable. Our approach differs both conceptually and methodically from method proposed by Jones et al. [24]. Methodically, we used Iterated F-Race [26] as an optimization algorithm and a more restricted architecture for the behavior trees. Conceptually, we focussed on showing that automatic modular design can cross the reality gap in a satisfactory

way, even when using different architectures. Furthermore, for the leaves of the behavior tree we used complex low-level behaviors instead of atomic actions.

7 Conclusions

AutoMoDe, automatic modular design, is an approach in which control software for robot swarms is automatically generated by assembling and fine-tuning preexisting modules. In previous articles, the control software architecture on which AutoMoDe operates was arbitrarily restricted to probabilistic finite state machine. In this article, we went beyond this restriction and we investigated the possibility of adopting behavior trees as a control software architecture. Behavior trees are appealing for a number of reasons. Compared to finite state machines, behavior trees offer greater expressiveness, implement the principle of two-way control transfers, and posses inherent modularity which allows the creation of a hierarchical structure, code reuse, and separation of concerns. Behavior trees are also easier to manipulate without compromising their integrity. This fact could be extremely useful when designing optimization algorithms based on iterative improvement.

We proposed a new instance of AutoMoDe called `Maple`, which fine-tunes and assembles preexisting modules into a behavior tree. To highlight its potential, we performed experiments in simulation and reality for two different missions: FORAGING and AGGREGATION. The results show that `Maple` performs similar to `Chocolate`—the state-of-the-art AutoMoDe method, which generates probabilistic finite state machines. They both cross the reality gap in a satisfactory way. `EvoStick`, which is an evolutionary robotics method, performs better then `Maple` and `Chocolate` in simulation, but significantly worse in reality.

Future work will focus on fully exploiting the potentials of behavior trees. This implies defining modules that are natively conceived to operate within a behavior tree—e.g., modules that properly return their state value (*success*, *running*, or *failure*) and therefore interact correctly with all possible control nodes. Moreover, we will define an ad-hoc optimization algorithm, possibly relying also on iterative improvement, that fully exploits the inherent modularity and hierarchical structure of behavior trees.

Acknowledgements. The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681872). Mauro Birattari acknowledges support from the Belgian *Fonds de la Recherche Scientifique* – FNRS.

References

1. Becroft, D., Bassett, J., Mejía, A., Rich, C., Sidner, C.L.: Aipaint: A sketch-based behavior tree authoring tool. In: Bulitko, V., Riedl, M.O. (eds.) Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-11. AAAI Press, Stanford, California, USA (2011)

2. Beni, G.: From swarm intelligence to swarm robotics. In: Swarm Robotics, SAB. LNCS, vol. 3342, pp. 1–9. Springer, Berlin, Germany (2004). https://doi.org/10.1007/978-3-540-30552-1_1
3. Birattari, M.: On the estimation of the expected performance of a metaheuristic on a class of instances. how many instances, how many runs? Tech. Rep. TR/IRIDIA/2004-01, IRIDIA, Université libre de Bruxelles, Belgium (2004)
4. Bozhinoski, D., Birattari, M.: Designing control software for robot swarms: Software engineering for the development of automatic design methods. In: ACM/IEEE 1st International Workshop on Robotics Software Engineering, RoSE. pp. 33–35. ACM, New York (2018)
5. Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. **7**(1), 1–41 (2013). https://doi.org/10.1007/s11721-012-0075-2
6. Brooks, R.: A robust layered control system for a mobile robot. IEEE Journal on Robotics and Automation **2**(1), 14–23 (1986)
7. Burridge, R.R., Rizzi, A.A., Koditschek, D.E.: Sequential composition of dynamically dexterous robot behaviors. The International Journal of Robotics Research **18**(6), 534–555 (1999)
8. Champandard, A.J.: Understanding behavior trees. `http://aigamedev.com/open/articles/bt-overview/` (2007)
9. Colledanchise, M., Ögren, P.: How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. IEEE Transactions on Robotics **33**(2), 372–389 (2017)
10. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: An introduction. `https://arxiv.org/abs/1709.00084` (2018)
11. Dorigo, M., Birattari, M., Brambilla, M.: Swarm robotics. Scholarpedia **9**(1), 1463 (2014). https://doi.org/10.4249/scholarpedia.1463
12. Duarte, M., Gomes, J., Costa, V., Oliveira, S.M., Christensen, A.L.: Hybrid control for a real swarm robotics system in an intruder detection task. In: Squillero, G., Burelli, P. (eds.) Applications of Evolutionary Computation, 19th European Conference, EvoApplications 2016. LNCS, vol. 9598, pp. 213–230. Springer, Cham, Switzerland (2016)
13. Floreano, D., Husbands, P., Nolfi, S.: Evolutionary robotics. Handbook of Robotics pp. 1423–1451 (2008)
14. Francesca, G., Birattari, M.: Automatic design of robot swarms: achievements and challenges. Front. Robot. AI **3**(29), 1–9 (2016). https://doi.org/10.3389/frobt.2016.00029
15. Francesca, G., Brambilla, M., Brutschy, A., Garattoni, L., Miletitch, R., Podevijn, G., Reina, A., Soleymani, T., Salvaro, M., Pinciroli, C., Birattari, M.: AutoMoDe-Chocolate: automatic design of control software for robot swarms. Swarm Intell. **9**(2/3), 125–152 (2015). https://doi.org/10.1007/s11721-015-0107-9
16. Francesca, G., Brambilla, M., Brutschy, A., Trianni, V., Birattari, M.: AutoMoDe: a novel approach to the automatic design of control software for robot swarms. Swarm Intell. **8**(2), 89–112 (2014). https://doi.org/10.1007/s11721-014-0092-4
17. Francesca, G., Brambilla, M., Trianni, V., Dorigo, M., Birattari, M.: Analysing an evolved robotic behaviour using a biological model of collegial decision making. In: Tom Ziemke, Christian Balkenius, J.H. (ed.) From Animals to Animats 12: 12th International Conference on Simulation of Adaptive Behavior, SAB. LNCS, vol. 7426, pp. 381–390. Springer, Berlin, Germany (2012)

18. Garattoni, L., Francesca, G., Brutschy, A., Pinciroli, C., Birattari, M.: Software infrastructure for e-puck (and TAM). Tech. Rep. TR/IRIDIA/2015-004, IRIDIA, Université libre de Bruxelles, Belgium (2015)
19. Garattoni, L., Birattari, M.: Swarm robotics. In: Webster, J. (ed.) Wiley Encyclopedia of Electrical and Electronics Engineering. John Wiley & Sons, Hoboken NJ (2016)
20. Gutiérrez, Á., Campo, A., Dorigo, M., Donate, J., Monasterio-Huelin, F., Magdalena, L.: Open e-puck range & bearing miniaturized board for local communication in swarm robotics. In: Kosuge, K. (ed.) IEEE International Conference on Robotics and Automation, ICRA. pp. 3111–3116. IEEE, Piscataway, NJ (2009)
21. Hasselmann, K., Ligot, A., Francesca, G., Birattari, M.: Reference models for AutoMoDe. Tech. Rep. TR/IRIDIA/2018-002, IRIDIA, Université libre de Bruxelles, Belgium (2018)
22. Hu, D., Gong, Y., Hannaford, B., Seibel, E.J.: Semi-autonomous simulated brain tumor ablation with RavenII surgical robot using behavior tree. In: Parker, L., et al. (eds.) IEEE International Conference on Robotics and Automation, ICRA. pp. 3868–3875. IEEE, Piscataway, NJ (2015)
23. Isla, D.: Handling complexity in the Halo 2 AI. In: Game Developers Conference. vol. 12 (2005)
24. Jones, S., Studley, M., Hauert, S., Winfield, A.: Evolving behaviour trees for swarm robotics. In: Distributed Autonomous Robotic Systems. pp. 487–501. Springer, Cham, Switzerland (2016). https://doi.org/10.1007/978-3-319-73008-0_34
25. Kuckling, J., Ligot, A., Bozhinoski, D., Birattari, M.: Behavior trees as a control architecture in the automatic design of robot swarms: Supplementary material. `http://iridia.ulb.ac.be/supp/IridiaSupp2018-004/index.html` (2018)
26. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
27. Marzinotto, A., Colledanchise, M., Smith, C., Ögren, P.: Towards a unified behavior trees framework for robot control. In: Xi, N., et al. (eds.) IEEE International Conference on Robotics and Automation, ICRA. pp. 5420–5427. IEEE, Piscataway, NJ (2014)
28. Menzel, R.: Spectral Sensitivity and Color Vision in Invertebrates, pp. 503–580. Springer, Berlin, Germany (1979)
29. Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.C., Floreano, D., Martinoli, A.: The e-puck, a robot designed for education in engineering. In: Gonçalves, P., Torres, P., Alves, C. (eds.) Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions. pp. 59–65. Instituto Politécnico de Castelo Branco, Portugal (2009)
30. Nehaniv, C.L., Dautenhahn, K.: Imitation in animals and artifacts. MIT press (2002)
31. Ögren, P.: Increasing modularity of UAV control systems using computer game behavior trees. In: Thienel, J., et al. (eds.) AIAA guidance, navigation, and control conference 2012. pp. 358–393. AIAA Meeting Papers (2012)
32. Perez, D., Nicolau, M., O'Neill, M., Brabazon, A.: Evolving behaviour trees for the Mario AI competition using grammatical evolution. In: Di Chio, C., et al. (eds.) Applications of Evolutionary Computation. LNCS, vol. 6624, pp. 123–132. Springer, Berlin, Germany (2011)
33. Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Birattari, M., Gambardella, L.,

Dorigo, M.: ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. Swarm Intell. **6**(4), 271–295 (2012)
34. Şahin, E.: Swarm robotics: From sources of inspiration to domains of application. In: Swarm Robotics, SAB. LNCS, vol. 3342, pp. 10–20. Springer, Berlin, Germany (2004)
35. Spears, W.M., Spears, D., Hamann, J.C., Heil, R.: Distributed, physics-based control of swarms of vehicles. Autonomous Robots **17**, 137–162 (2004)
36. Stranieri, A., Turgut, A.E., Salvaro, M., Garattoni, L., Francesca, G., Reina, A., Dorigo, M., Birattari, M.: IRIDIA's arena tracking system. Tech. Rep. TR/IRIDIA/2013-013, IRIDIA, Université libre de Bruxelles, Belgium (2013)
37. Trianni, V.: Evolutionary Swarm Robotics. Springer, Berlin, Germany (2008)