AutoMoDe-Chocolate: a Method for the Automatic Design of Robot Swarms that Outperforms Humans

G. Francesca, M. Brambilla, A. Brutschy, L. Garattoni, R. Miletitch, G. Podevin, A. Reina, T. Soleymani, M. Salvaro, C. Pinchiroli,
F. Mascia, V. Trianni, and M. Birattari

IRIDIA – Technical Report Series
TR/IRIDIA/2014-015
November 2014
AutoMoDe-Chocolate: a method for the automatic design of robot swarms that outperforms humans

Gianpiero Francesca · Manuele Brambilla · Arne Brutschy · Lorenzo Garattoni · Roman Miletitch · Gaëtan Podevijn · Andreagiovanni Reina · Touraj Soleymani · Mattia Salvaro · Carlo Pinciroli · Franco Mascia · Vito Trianni · Mauro Birattari

Received: date / Accepted: date

Abstract We introduce AutoMoDe-Chocolate (hereafter Chocolate), an automatic method to design control software for robot swarms. Chocolate is an improved version of AutoMoDe-Vanilla (hereafter Vanilla). Analogously to Vanilla, Chocolate generates control software in the form of a finite state machine by assembling and fine-tuning preexisting parametric modules. The only difference between Vanilla and Chocolate is the optimization algorithm adopted to search the space of the possible designs—while Vanilla adopts F-Race, Chocolate adopts Iterated F-Race.

The results of an empirical analysis performed on five swarm robotics tasks show that the control software designed by Chocolate outperforms both the one designed by Vanilla and the one designed by a human expert. Chocolate is the first automatic design method for robot swarms that is shown to outperform a human.

Keywords swarm robotics · automatic design · AutoMoDe

1 Introduction

In this paper we introduce AutoMoDe-Chocolate, an automatic method to design control software for robot swarms.

The design of robot swarms is challenging due to the complex relation existing between the individual behavior of each robot and the resulting swarm-level properties: The requirements of the swarm are naturally expressed at the collective level by stating the characteristics of the desired collective behavior that the swarm should exhibit. Nonetheless, the designer must eventually define what the individual robot should do, so that the desired collective behavior is achieved (Dorigo et al, 2014).

Presently, no general approach exists to derive the individual behavior of the robots from a desired collective behavior. Typically, robot swarms are designed
manually via a trial-and-error process: the designer implements, tests, and modifies the behavior of the individual robots until the desired collective behavior is obtained. This design process completely relies on the intuition and skills of the designer. More importantly, this process lacks the repeatability and consistency that characterize engineering practice.

Automatic design is an appealing alternative to the manual design process described above. In an automatic method, the design problem is cast into an optimization problem. An automatic method produces instances of control software that conform to a predefined parametric architecture. The free parameters of the architecture are tuned via an optimization algorithm.

Several studies have shown that effective control software for robot swarms can be produced via an optimization process (e.g., see Trianni and Dorigo, 2006; Trianni and Nolfi, 2009; Groß and Dorigo, 2009). However, these studies owe their success to task-specific solutions, leaving the problem of creating a general methodology still open (Trianni and Nolfi, 2011).

The focus of our research is on developing a truly general-purpose, automatic design method capable of producing control software for robot swarms. In particular, by general we mean that the method must prove effective for a sufficiently large class of tasks, without requiring task-specific modifications.

As a first step in this direction, we introduced AutoMoDe in Francesca et al (2014a). AutoMoDe designs control software in the form of a probabilistic finite state machine, by combining and fine-tuning preexisting parametric modules. We implemented a first version of AutoMoDe called AutoMoDe-Vanilla (hereafter Vanilla). Vanilla designs control software for a specific version of the e-puck robot (Mondada et al, 2009), as formally described by a reference model (Francesca et al, 2014a). We compared Vanilla with EvoStick, a design method that uses an evolutionary algorithm to optimize a neural network. To ensure a fair and meaningful comparison, EvoStick adopts the same reference model adopted by Vanilla. The comparison was based on two classical swarm robotics tasks: aggregation and foraging. Our results show that on both the tasks Vanilla outperforms EvoStick. For more details, we refer the interested reader to Francesca et al (2014a).

In Francesca et al (2014b), we performed a second comparison on five new tasks: shelter with constrained access, largest covering network, coverage with forbidden areas, surface and perimeter coverage, and aggregation with ambient cues. This second comparison involved the exact same versions of Vanilla and EvoStick as they were defined in Francesca et al (2014a): the two methods have not undergone any modification to adapt them to the new tasks. In particular, the modules on which Vanilla operates are the same defined in Francesca et al (2014a) and used there to produce control software for aggregation and foraging. The five tasks were defined by researchers that had knowledge of Vanilla and EvoStick’s reference model, but that were unaware of the internals of Vanilla and EvoStick. This ensures that the definition of the tasks is neutral and no a priori advantage is granted to any method. In the analysis, we included also two manual design methods: C-Human and U-Human. In C-Human, the human designer is constrained to implement control software by combining the same modules on which Vanilla operates. In U-Human, the human designer is unconstrained and implements the control software without any restriction on the structure of the solution. U-Human is the standard way in which control software for robot swarms is currently produced (Brambilla et al, 2013).
The results of the empirical analysis presented in Francesca et al (2014b) are summarized in Figure 1. These results confirm those obtained in Francesca et al (2014a): also on the five new tasks, Vanilla is able to design control software that performs better than the one designed by EvoStick. Concerning the comparison between Vanilla and manual design, Vanilla outperforms U-Human, that is, the human designer that implements control software without constraints on the solution. However, Vanilla performs worse than C-Human. As Vanilla and C-Human operate on the same set of modules, the difference in performance is to be ascribed to the mechanism adopted by Vanilla to combine and fine-tune the modules: the optimization algorithm. In the light of this conclusion, in Francesca et al (2014b) we argue that Vanilla could be improved by adopting a better optimization algorithm.

AutoMoDe-Chocolate (hereafter Chocolate) is an improved version of Vanilla that is based on a new optimization algorithm. To assess Chocolate, we perform a new empirical analysis under the same conditions defined in Francesca et al (2014b). The goal of this analysis is to confirm the following working hypotheses: (i) the weak point of Vanilla is its optimization algorithm; (ii) by adopting a more advanced optimization algorithm, Chocolate improves over Vanilla; and (iii) the improvement is such that Chocolate outperforms C-Human. The results of the analysis, which are reported in Section 5, confirm our hypotheses.

The significance of the contribution made in this paper lies in the fact that Chocolate is the first automatic method that is shown capable of producing better control software for robot swarms than a human designer.

2 Related work

Automatic design is a promising approach to produce control software for robot swarms. In the following we review the most relevant and recent literature on the topic. For a comprehensive survey of the swarm robotics literature, we refer the reader to Brambilla et al (2013).

A number of studies tackled the problem of designing control software in the form of a neural network using evolutionary algorithms. Typically, the focus of these studies is on a specific task rather than on the development of a general-purpose design method. Trianni and Dorigo (2006) developed a hole avoidance behavior for a group of four robots, Trianni and Nolfi (2009) developed a synchronization behavior, Groß and Dorigo (2009) developed a collective transport...
behavior. We refer the reader to Trianni and Nolfi (2011) for further references on the evolutionary approach.

More recently, Gauci et al (2014) focused on a clustering behavior. The automatic design method produces control software in the form of a simple lookup table that maps the reading of a single three-valued sensor into the desired wheel velocity. The parameters of the lookup table are obtained through an evolutionary algorithm.

Francesca et al (2014a) introduced AutoMoDe, an automatic design method that produces control software by combining and instantiating preexisting modules in the form of a probabilistic finite state machine. Vanilla, a first implementation of AutoMoDe that targets the e-puck robot, has been shown capable of designing control software for seven different tasks: aggregation, foraging, shelter with constrained access, largest covering network, coverage with forbidden areas, surface and perimeter coverage, and aggregation with ambient cues (Francesca et al, 2014a,b).

Two other research directions are relevant and deserve being mentioned here: on-line adaptation and design via formal methods. Although it often concerns only few parameters, on-line adaptation can be considered as a sort of automatic design performed at runtime. In contrast, formal methods, despite being a manual design approach, are relevant in the context of automatic design because they are structured procedures that could possibly be automatized.

Two recent works constitute significant examples of on-line adaptation in swarm robotics. Di Mario and Martinoli (2014) applied a particle swarm optimization algorithm for on-line adaptation of control software that performs obstacle avoidance. The adaptation mechanism has been demonstrated on a swarm of up to eight Khepera III robots. Bredeche et al (2012) introduced an approach based on an evolutionary algorithm that allows a swarm of e-puck robots to adapt to environmental changes in a survival and reproduction task.

Two recent works on formal methods for robot swarms are particularly relevant. Lopes et al (2014) introduced an approach based on supervisory control theory. To demonstrate their approach, the authors designed a segregation behavior for swarms of e-pucks and kilobots (Rubenstein et al, 2014). Brambilla et al (2014) introduced an approach based on prescriptive modeling and model checking. To demonstrate their approach, the authors developed control software for solving aggregation and foraging with a swarm of e-puck robots.

3 Chocolate

Chocolate is an improved version of Vanilla, the first version of AutoMoDe that was introduced in Francesca et al (2014a). Chocolate designs control software for the e-puck, the same robot for which Vanilla was conceived. More specifically, Chocolate adopts the same reference model of the e-puck used by Vanilla. Moreover, Chocolate and Vanilla generate control software by operating on the same set of modules. The only difference between Chocolate and Vanilla is the optimization algorithm used to explore the design space.

For sake of completeness, in this section we provide a self-contained description of Chocolate.
Chocolate designs control software for a specific version of the e-puck robot, whose capabilities are formally defined via the reference model introduced by Francesca et al (2014a). The version of the e-puck robot and the reference model for which Chocolate designs control software are the same of Vanilla. With the goal of providing the reader with a self-contained definition of Chocolate, we describe in the following the e-puck robot and we detail the adopted reference model.

The e-puck maneuvers by actuating its two wheels, which constitute a differential steering system (Mondada et al, 2009). The version of the e-puck adopted in our research is showed in Figure 2. This version of the e-puck is equipped with 8 infrared transceivers, 3 ground sensors, and a range-and-bearing board. The infrared transceivers are placed around the body of the e-puck and are used as light and proximity sensors: they allow the e-puck to measure the intensity of light and sense the presence of nearby obstacles. The ground sensors are placed under the front of the e-puck and can measure the reflectance of the floor, which allows the e-puck to distinguish at least three levels of gray. The range-and-bearing board (Gutiérrez et al, 2009) allows the e-puck to perceive the presence of other e-pucks in a 0.70 m range. For each perceived e-puck, the range-and-bearing board computes the distance (range) and the relative angle (bearing).

The described capabilities of the e-puck are formalized in a reference model, which abstracts sensors and actuators by defining the input and the output variables that are made available to the control software at each control step. Sensors are defined as input variables and can be only read by the control software. Actuators are defined as output variables and can be only written by the control software. Input and output variables are updated with a period of 100 ms. The reference model is summarized in Table 1. According to this reference model, the

---

1 The range-and-bearing board also allows the e-pucks to exchange messages. However, this functionality is not included in the reference model defined in Francesca et al (2014a).
Table 1 Reference model

<table>
<thead>
<tr>
<th>Input Variable</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prox$_i$ $\in{1,2,...,8}$</td>
<td>$[0,1]$</td>
<td>reading of the $i$-th proximity sensor</td>
</tr>
<tr>
<td>light$_i$ $\in{1,2,...,8}$</td>
<td>$[0,1]$</td>
<td>reading of the $i$-th light sensor</td>
</tr>
<tr>
<td>gnd$_j$ $\in{1,2,3}$</td>
<td>{black, gray, white}</td>
<td>floor color read by $j$-th ground sensor</td>
</tr>
<tr>
<td>$n$</td>
<td>${0, \ldots, 20}$</td>
<td>number of neighboring e-pucks</td>
</tr>
<tr>
<td>$r_m$ $\in{1,2,...,n}$</td>
<td>$[0,0.70]$</td>
<td>distance of $m$-th neighboring e-puck</td>
</tr>
<tr>
<td>$\angle b_m$ $\in{1,2,...,n}$</td>
<td>$[0,2\pi]$</td>
<td>rad angle of $m$-th neighboring e-puck</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Variable</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_k$ $\in{l,r}$</td>
<td>$[-0.16, 0.16]$</td>
<td>m/s target linear wheel velocity</td>
</tr>
</tbody>
</table>

Period of the control cycle: 100 ms

Reading of a proximity sensor $i$ is stored in the variable $prox_i$ that has values between 0 and 1. When sensor $i$ does not perceive any obstacle in a 0.03 m range, $prox_i = 0$; while when sensor $i$ perceives an obstacle closer than 0.01 m, $prox_i = 1$. Similarly, the reading of a light sensor $i$ is stored in the variable $light_i$ that has values between 0, when no light source is perceived, and 1, when the sensor $i$ saturates. The readings of the three ground sensors are stored in the variables $gnd_1$, $gnd_2$ and $gnd_3$. These variables can take three different values: black, gray and white. The e-puck uses the range-and-bearing board to perceive other e-pucks in its neighborhood. The variable $n$ stores the number of the neighboring e-pucks. For each neighboring e-puck $m \in \{1, 2, \ldots, n\}$, the variables $r_m$ and $\angle b_m$ indicate the range and the bearing, respectively. The wheel actuators are operated by the control software through the variables $v_l$ and $v_r$, in which the control software writes the target linear velocity for the left and right wheel, respectively. The linear wheel velocity ranges between $-0.16$ m/s and $0.16$ m/s.

3.2 Modules and probabilistic finite state machines

Chocolate produces robot control software by assembling preexisting modules into a probabilistic finite state machine. The preexisting modules operate on the variables defined in the reference model given in Section 3.1. Modules might have parameters that regulate their internal functioning. The parameters, along with the topology of the probabilistic finite state machine, are optimized in order to maximize a task-dependent performance measure. The way in which modules are assembled and optimized by Chocolate is described in the following Section 3.3. The remaining of the present section is devoted to a description of the functioning of the probabilistic finite state machine and of the modules.

Chocolate assembles two kind of modules: behaviors and transitions. A behavior is an activity that the robot can perform, while a transition is a criterion to regulate the change of behavior in response to a particular condition or event experienced by the robot. In practice, a behavior is a parametric procedure that sets the output variables defined in the reference model on the basis of the value of (a subset of) the input variables. A transition is a parametric procedure that returns either true or false on the basis of the value of (a subset of) the input variables.
In the parlance of probabilistic finite state machines, states and edges are instances of behaviors and transitions, respectively. More precisely, a state (edge) is an instance of a behavior (transition) in which the parameters, if any, are given a valid value. Different states (edges) might be instances of the same behavior (transition), possibly with different values of the parameters.

An execution of the control software is a series of control steps of 100 ms each. At any given control step, the probabilistic finite state machine is in one and only one state, which we refer to as the active state. The instance of the behavior associated with the active state is executed. That is, output variables are set, on the basis of the input variables, as prescribed by the behavior. Subsequently, if at least one outgoing transition returns true, the control software changes state: one transition among the ones that returned true is randomly selected and the state pointed by the selected transition becomes the active state for the following control step. If no transition returns true, the active state remains unchanged. The execution of the control software then moves on to the following control step.

In Chocolate, twelve modules are available for being assembled into a probabilistic finite state machine: six behaviors and six transitions. These twelve modules are exactly the same ones employed in Vanilla. We provide here a succinct description of these modules and we refer the reader to Vanilla’s original paper for the details (Francesca et al, 2014a). In this description, as in Vanilla’s original paper, we use Greek letters to indicate the parameters of the modules.

### 3.2.1 Behaviors

#### exploration

<table>
<thead>
<tr>
<th>Parameters: $\tau \in {1, \ldots, 100}$</th>
<th>Variables: $prox; v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>the robot goes straight; if it hits an obstacle, it rotates on itself for a random number of control steps chosen in the interval $[0, \tau]$</td>
<td></td>
</tr>
</tbody>
</table>

#### stop

<table>
<thead>
<tr>
<th>Parameters: none</th>
<th>Variables: none</th>
</tr>
</thead>
<tbody>
<tr>
<td>the robot stays still</td>
<td></td>
</tr>
</tbody>
</table>

#### phototaxis

<table>
<thead>
<tr>
<th>Parameters: none</th>
<th>Variables: $prox; light; v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>the robot moves towards the light source, if perceived; otherwise it moves straight. This behavior embeds obstacle avoidance</td>
<td></td>
</tr>
</tbody>
</table>

#### anti-phototaxis

<table>
<thead>
<tr>
<th>Parameters: none</th>
<th>Variables: $prox; light; v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>the robot moves away from the light source, if perceived; otherwise it moves straight. This behavior embeds obstacle avoidance</td>
<td></td>
</tr>
</tbody>
</table>
attraction

\[
\text{parameters: } \alpha \in [1, 5] \quad \text{variables: } \text{prox}; n; r; \angle b; v
\]

the robot moves towards the neighboring robots. The direction vector \( w \) is computed on the basis of the range and bearing of the neighboring robots:
\[
w = \sum_{m=1}^{n} (\alpha_{r_m}, \angle b_m).
\]
This behavior embeds obstacle avoidance.

repulsion

\[
\text{parameters: } \alpha \in [1, 5] \quad \text{variables: } \text{prox}; n; r; \angle b; v
\]

the robot moves away from the neighboring robots. The direction vector \( w \) is computed on the basis of the range and bearing of the neighboring robots:
\[
w = -\sum_{m=1}^{n} (\alpha_{r_m}, \angle b_m).
\]
This behavior embeds obstacle avoidance.

3.2.2 Transitions

black-floor

\[
\text{parameters: } \beta \in [0, 1] \quad \text{variables: } \text{gnd}
\]
true with probability \( \beta \) if any of the ground sensors reads black; false otherwise

gray-floor

\[
\text{parameters: } \beta \in [0, 1] \quad \text{variables: } \text{gnd}
\]
true with probability \( \beta \) if any of the ground sensors reads gray; false otherwise

white-floor

\[
\text{parameters: } \beta \in [0, 1] \quad \text{variables: } \text{gnd}
\]
true with probability \( \beta \) if any of the ground sensors reads white; false otherwise

neighbors-count

\[
\text{parameters: } \eta \in [0, 20]; \xi \in \{0, 1, \ldots, 10\} \quad \text{variables: } n
\]
true according to the probability distribution
\[
z(n) = \frac{1}{1+e^{\eta(\xi-n)}}
\]

inverted-neighbors-count

\[
\text{parameters: } \eta \in [0, 20]; \xi \in \{0, \ldots, 10\} \quad \text{variables: } n
\]
true according to the probability distribution \( 1 - z(n) \), where \( z(n) \) is defined as in neighbors-count

fixed-probability

\[
\text{parameters: } \beta \in [0, 1] \quad \text{variables: } \text{none}
\]
true with probability \( \beta \)
Figure 3 depicts an example of probabilistic finite state machine that can be generated with the modules described above. Specifically, the probabilistic finite state machine represented here has been produced by Vanilla as a solution to a task in which e-puck robots are required to aggregate on a black region (Francesca et al, 2014a).

3.3 Optimization algorithm

In Chocolate, as in Vanilla, the automatic generation of control software is cast into an optimization problem. 

A feasible solution of the optimization problem is an instance of control software—that is, a specific probabilistic finite state machine assembled using the modules described in Section 3.2, together with specific values of their parameters. More precisely, the space of feasible solutions searched by Chocolate is the same one searched by Vanilla: the two methods search the space of the probabilistic finite state machines that comprise up to four states and up to four outgoing edges from each state; the behaviors and the transitions to be associated with states and edges, respectively, are sampled with replacement from the modules described in Section 3.2; the valid range of the parameters of each module is given in Section 3.2.

The goal of the optimization is to maximize the expected value of a task-specific performance measure; where the expectation is taken with respect to the initial conditions and the contingencies of task execution. Each different initial condition starting from which the task has to be performed is reproduced through a different test case on which solutions are evaluated. The contingencies of task execution are accounted for through realistic computer-based simulations that reproduce sensor and actuator noise. Specifically, in Chocolate as in Vanilla, a solution is evaluated on a test case by means of ARGoS (Pinciroli et al, 2012), a physics-based simulator of swarm robotics systems that includes a realistic model of the e-puck robot.

As mentioned above, the only difference between Chocolate and Vanilla is the optimization algorithm adopted. Chocolate adopts Iterated F-Race (Balaprakash et al, 2007; Birattari et al, 2010; López-Ibáñez et al, 2011), an algorithm for auto-
matic configuration originally devised to fine-tune the parameters of metaheuristics. Iterated F-Race is based on F-Race (Brattari et al, 2002; Birattari, 2009), the optimization algorithm adopted in Vanilla.

In F-Race, a set of candidate solutions are sequentially evaluated over different test cases in a process that is reminiscent of a race. The aim of the process is to select the best candidate solution. The set of candidate solutions is sampled in a uniformly random way from the space of feasible solutions. The F-Race algorithm comprises a series of steps. At each step, a different test case is sampled and is used to evaluate the candidate solution. At the end of each step, a Friedman test is performed on the basis of the results obtained by the candidate solutions on the test cases sampled so far. All candidate solutions that appear to perform significantly worse than at least another one are dropped from the set of candidate solutions and are not further evaluated in the subsequent steps. The process stops when either a single candidate remains or when a predefined budget of evaluations have been performed. By discarding as early as possible the candidates that are statistically dominated by at least another candidate, the evaluation process implemented by the F-Race algorithm allows for a rational and efficient use of the available evaluation budget (Birattari, 2009).

In Iterated F-Race, the optimization process goes through a series of iterations each of which is an execution of the F-Race algorithm. In the first iteration, an initial set of candidate solutions is generated by sampling the space of feasible solutions in a uniformly random way. The initial candidates undergo a first execution of the F-Race algorithm. When the F-Race algorithm terminates, the surviving solutions—that is, the candidate solutions that have not been discarded—are used as a seed to generate a new set of candidate solutions on which the following iteration will operate. The new set of candidates is obtained by sampling the space of feasible solutions according to a distribution that gives a higher probability of being selected to solutions that are close to the surviving solutions. See López-Ibáñez et al (2011) for the details. The new set of candidates undergo a further execution of the F-Race algorithm. The process stops when a predefined budget of evaluations have been performed.

The implementation of Iterated F-Race that is adopted in Chocolate is the one provided by the irace package (López-Ibáñez et al, 2011) for R (R Core Team, 2014). The irace package contains implementations of a number of racing procedures including also the version of F-Race that is adopted in Vanilla (Francesca et al, 2014a). Chocolate and Vanilla use the default parameters of Iterated F-Race and F-Race, respectively. Moreover, Chocolate, as Vanilla, samples the design space using the built-in sampling procedure provided by irace (López-Ibáñez et al, 2011).

4 Experimental protocol

To evaluate Chocolate, we adopt an experimental protocol that shares its main features with the one defined in Francesca et al (2014b). The only differences concern (i) the way in which the tasks are defined/selected and (ii) the design methods under analysis.
4.1 Tasks

The analysis is performed on five swarm robotics tasks. The tasks were defined by five swarm robotics experts in Francesca et al (2014b). In the definition of the tasks, the experts were kept unaware of the design methods included in the experimental analysis, in order to avoid any influence in the experts’ choices that could favor one method over the others. Experts were provided with the reference model described in Section 3.1 and were asked to define tasks that, according to their judgment, could be performed by a swarm of 20 e-puck robots conforming to the given reference model.

The experts were given a list of constraints that the tasks must satisfy: The time available to the e-pucks for performing a task is $T = 120 \text{s}$. The e-pucks operate in a dodecagonal area of $4.91 \text{m}^2$ surrounded by walls. The floor of the arena is gray. Up to three circular or rectangular patches might be present on the floor. The patches might be either white or black. The diameter of the circular patches and the sides of the rectangular patches cannot exceed 0.6 m. The experimental setup might include a light source placed outside the south side of the arena. Up to 5 obstacles might be present in the arena. Obstacles are wooden cuboids of size $0.05 \text{m} \times 0.05 \text{m} \times L$, where $L$ is in the range $[0.05, 0.80] \text{m}$.

As part of the task definition, the experts were asked to define a performance measure that can be used to assess task execution. The performance measure should be computable on the basis of the position and orientation of the e-pucks, evaluated every 100 ms.

In this paper, we use the five tasks defined in Francesca et al (2014b), rather than repeating the task definition process adopted therein. The tasks we consider are: SCA – shelter with constrained access, LCN – largest covering network, CFA – coverage with forbidden areas, SPC – surface and perimeter coverage, and AAC – aggregation with ambient cues. For each task, Figure 4 shows the corresponding experimental arena. A description of the tasks is given in the following:

**SCA – shelter with constrained access.** The arena contains a rectangular white region of $0.15 \text{m} \times 0.6 \text{m}$. This region is closed on three sides by obstacles; only the south side is open for the e-pucks to enter. In the arena, there are also two black circular patches, positioned aside the white region. The two circular patches have the same diameter of 0.6 m. The setup includes also a light source placed on the south side of the arena. The task for the e-pucks is to aggregate on the white region: the shelter. The e-pucks can use the light source and the black circular patches to orientate themselves in the arena. The performance measure is defined in terms of an objective function to be maximized: $F_{SCA} = \sum_{t=1}^{T} N(t)$, where $N(t)$ is the number of e-pucks in the shelter at time $t$ and $T$ is the duration of the experiment.

**LCN – largest covering network.** The arena does not contain any obstacle, floor patch or light source. The e-pucks are required to create a connected network that
SCA – shelter with constrained access
robots must aggregate in the white region, the shelter

LCN – largest covering network
robots must create a connected network that covers the largest area possible

CFA – coverage with forbidden areas
robots must cover all the arena except the forbidden black regions

SPC – surface and perimeter coverage
robots must cover the area of the white square and the perimeter of the black circle

AAC – aggregation with ambient cues
robots must aggregate on the black circle

Fig. 4 Overhead shots of the arenas used for the five tasks considered in the experimental analysis. The pictures show also the 20 e-puck robots
covers the largest area possible. Each e-puck covers a circular area whose radius is 0.35 m. Two e-pucks are considered to be connected if their distance is less than 0.25 m. The performance measure is defined in terms of an objective function to be maximized: $F_{LCN} = A_{C(T)}$, where $C(T)$ is the largest network of connected e-pucks at the end $T$ of an experiment and $A_{C(T)}$ is the area covered by the network $C(T)$.

CFA – coverage with forbidden areas. The arena contains three circular black regions, each with a diameter of 0.6 m. The e-pucks are required to cover the entire arena, avoiding the forbidden areas denoted by the black floor. The performance measure is defined in terms of an objective function to be minimized: $F_{CFA} = E[d(T)]$, where $E[d(T)]$ is the expected distance, at the end $T$ of an experiment, between a generic point of the arena and the closest e-puck that is not in the forbidden area. This objective function is measured in meters.

SPC – surface and perimeter coverage. The arena contains a circular black region that has a diameter of 0.6 m and a square white region that has sides of 0.6 m. The e-pucks are required to aggregate on the perimeter of the black circle and to cover the area of the white square. The performance measure is defined in terms of an objective function to be minimized: $F_{SPC} = c_a E[d_a(T)] + c_p E[d_p(T)]$, where $c_a = 0.08$, $c_p = 0.06$, $E[d_a(T)]$ is the expected distance, at the end $T$ of an experiment, between a generic point in the square region and the closest e-puck that is in the square region, $E[d_p(T)]$ is the expected distance between a generic point on the circumference of the circular region and the closest e-puck that intersects the circumference itself.

AAC – aggregation with ambient cues. The arena contains two circular regions, one black and one white, each with a diameter of 0.6 m. The black region is placed closer to the light source, which is on the south side of the arena. The e-pucks have to aggregate on the black region and can use the light and the white region to orientate themselves. The performance measure is defined in terms of an objective function to be maximized: $F_{AAC} = \sum_{t=1}^{T} N(t)$, where $N(t)$ is the number of e-pucks on the black region at time $t$.

The tasks described above are instances of the tasks that one can conceive if asked to devise a task that (i) satisfies the environmental constraints given above and (ii) can be performed by a swarm of 20 robots described by the reference model of Section 3.1. From an abstract standpoint, the environmental constraints and the reference model define the class of swarm robotics tasks that are relevant in the present study. In this sense, the definition of these tasks, as it has been performed by the experts, can be intended as a sampling procedure over the space of these tasks.

4.2 Design methods under analysis

We compare Chocolate, Vanilla, and C-Human, which is the best performing design method among those studied in Francesca et al (2014b).
The three design methods share a number of key characteristics: (i) they all produce robot control software in the form of a probabilistic finite state machine; (ii) they operate on the same set of modules; and (iii) they all adopt the same simulator to compare and select candidate designs.

**Chocolate** is described in Section 3. **Vanilla** was introduced in Francesca et al (2014a). The only difference between **Chocolate** and **Vanilla** is the optimization algorithm adopted: **Chocolate** adopts Iterated F-Race and **Vanilla** adopts F-Race. **C-Human** is a manual design method in which the control software is generated by a human expert. **C-Human** is described in Francesca et al (2014b): The role of the human experts is played by the same five researcher that defined the tasks—see Section 4.1. Each expert works on a task defined by another expert. Within the design process, the expert uses a simple application that generates a probabilistic finite state machine out of a high-level description that specifies which modules should be included, how they should be connected, and what value should be assigned to their parameters. The expert employs ARGoS to simulate the resulting robot swarm and to obtain the evaluation of the task-dependent performance measure.

In the protocol adopted in Francesca et al (2014b), two further design methods were included: **EvoStick** and **U-Human**. They have been excluded from the analysis presented in this paper because they were outperformed by **C-Human** and **Vanilla**.

### 4.3 Setup of the design process

The three design methods under analysis—**C-Human**, **Chocolate**, and **Vanilla**—are tested under the same conditions and are provided with the same resources:

**Same platform.** The methods all target the same robotic platform: the specific version of the e-puck formally defined by the reference model given in Section 3.1.

**Same simulator.** All methods employ ARGoS as a simulation software to evaluate design candidates.

**Same performance measure.** The methods all base the evaluation of a design candidate on the same task-specific performance measure.

**Same resources.** To design the control software, the three methods are given a comparable amount of time. **C-Human** is given four hours per task. Time starts when the human designer receives the description of the task. **Chocolate** and **Vanilla** are given a budget of 200,000 executions of ARGoS per task. **Chocolate** and **Vanilla** are executed on a computer cluster that comprises 400 opteron6272 cores. Under this setting, **Chocolate** and **Vanilla** are able to complete a design session in approximately 2 hours and 20 minutes, wall-clock time.

### 4.4 Experiments with the e-pucks

The instances of control software produced by **Chocolate**, **Vanilla**, and **C-Human** for each task are assessed via experiments with a swarm of 20 e-puck robots.

Although we have already performed an assessment of the control software generated by **Vanilla** and **C-Human** on the five tasks considered (Francesca et al, 2014b), we repeat these experiments. This allows us to compare the three methods.
under the exact same experimental conditions. It would be indeed practically impossible to reproduce the same conditions under which the experiments presented in Francesca et al. (2014b) were performed: Robots, batteries, and light sources have been subject to wear and their properties have possibly changed. Moreover, we have updated the firmware of the e-pucks and the software we use to manage the experiments and to track the position of the e-pucks over time. Finally, the arena has been disassembled and later reassembled in a different position.

The experimental session adopts a hands-off approach that reduces human intervention to a bare minimum. The control software is directly cross-compiled by the ARGoS simulator and it is installed on each e-puck of the swarm without any modification. To reduce the risk that the negative effects of battery discharge and other environmental contingencies affect one method more than another, the order of the experiments is randomly generated so that experiments with the control software produced by \textit{Chocolate}, \textit{Vanilla} and \textit{C-Human} are interleaved. The initial position of the e-pucks is randomly generated by placing the e-pucks in known positions and then performing the exploration behavior for 20 seconds.

To compute the task-dependent performance measure we use a tracking system (Stranieri et al, 2013) that gathers data via a ceiling-mounted camera. The tracking system logs position and orientation of each e-puck every 100 ms.

4.5 Objectives of the analysis and statistical tools

The main objective of the analysis is to compare the three design methods. In particular, we wish to confirm our working hypotheses: (i) the weak point of \textit{Vanilla} is its optimization algorithm; (ii) by adopting a more advanced optimization algorithm, \textit{Chocolate} improves over \textit{Vanilla}; and (iii) the improvement is such that \textit{Chocolate} outperforms \textit{C-Human}.

As discussed in Section 4.1, the selected tasks can be intended as a sample extracted from a class of tasks. As such, these tasks allow one to draw conclusions that generalize, in a statistical sense, to the class of tasks from which they have been sampled. For this reason, we concentrate our attention on the aggregate performance of the methods over the tasks considered. For the sake of completeness, we report also a boxplot of the per-task performance and the results obtained in simulation by the control software produced by the methods under analysis. Nonetheless, the focus of our study remains the aggregate analysis.

For each task, we perform 30 independent experiments: 10 for the control software instance generated by each of the three methods under analysis. We analyze the results using the Friedman test (Conover, 1999), with the task as a blocking factor. As the Friedman test is a rank-based non-parametric test, it does not require scaling the performance measure computed for each of the tasks nor formulating any restrictive hypothesis on the underlying distribution of the different performance measures. This test requires only to convert the objective functions of all tasks into the objective functions of the equivalent minimization problems. Given the rank-based nature of the Friedman test, this operation is trivial: it can be performed via any function that inverts the rank order. Specifically, to obtain a minimization problem from a maximization one, we use as objective function the inverse of the original one.
Concerning the per-task results of the three design methods, we present five notched box-and-whisker boxplots: one for each of the tasks. A notched box-and-whisker boxplot gives a visual representation of a sample. The horizontal thick line denotes the median of the sample. The lower and upper sides of the box are called upper and lower hinges and represent the 25-th and 75-th percentile 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—whatever is smaller. The lower whisker is defined similarly. Small circles represent outliers (if any), that is, observations that fall beyond the whiskers. 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. In the per-task boxplots, we include also the results obtained in simulation in order to appraise the impact of the reality gap on the three design methods. Results obtained in robot experiments are represented by wide boxes and those obtained in simulation by narrow boxes.

As with the experiments with the e-puck robots, also in simulation we perform 10 independent experiments for the control software instance generated by each of the three design methods under analysis.

5 Results

We present the results for each of the five task in Section 5.1 and the aggregate analysis in Section 5.2. The complete set of experimental data, along with the videos of all the experiments, is available as supplementary material in Francesca et al (2014c).

5.1 Per-task results

We report in the following the results obtained by the three methods under analysis on each of the five tasks.

**SCA – shelter with constrained access.** The results are reported in Figure 5. A visual inspection of the boxplot shows that the control software instance designed by Chocolate performs better than the ones designed by Vanilla and C-Human. The differences in performance between Chocolate and C-Human and between Chocolate and Vanilla are significant: the corresponding notches do not overlap. Moreover, C-Human performs significantly better than Vanilla. Regarding the difference between simulation and reality, Chocolate and C-Human appear to overcome the reality gap successfully: they have similar performance in simulation and in reality. On the contrary, Vanilla shows a significant mismatch.

**LCN – largest covering network.** The results are reported in Figure 6. A visual inspection of the boxplot shows that Chocolate and C-Human have qualitatively the same performance. On the other hand, Vanilla performs significantly worse than both Chocolate and C-Human. For what concerns the effects of the reality gap, all three design methods present a rather noticeable difference between simulation and reality. C-Human displays the smallest mismatch. The mismatch between the
Fig. 5  SCA – shelter with constrained access: results of the analysis. Chocolate performs significantly better than Vanilla and C-Human.

Fig. 6  LCN – largest covering network: results of the analysis. Chocolate and C-Human have similar performance. They both outperform Vanilla.

Performance in simulation and reality is possibly due to the fact that, to solve this task, the e-pucks rely on their ability to measure the distance of the neighboring robots. The measurement of the distance is obtained via the range-and-bearing board, which is imprecise and highly dependent on uncontrolled factors such as battery levels and light conditions.

CFA – coverage with forbidden areas. The results are reported in Figure 7. A visual inspection of the boxplot shows that Chocolate and C-Human have similar performance. Vanilla is slightly worse. Differences are small: the differences between the medians are all within a range of less that two centimeters.
effects of the reality gap, the three methods present a similar difference between simulation and reality. The flattening of the results with small differences that have no practical implications is possibly due to the imprecision of the distances measured via the range-and-bearing board.

**SPC – surface and perimeter coverage.** The results are reported in Figure 8. A visual inspection of the boxplot shows that the median performance recorder for Chocolate is better than the one recorded for C-Human and Vanilla. The difference between Chocolate and Vanilla is significant. Concerning the performance difference between simulation and robot experiments, all three the methods show
some mismatch. Like in the case of SCA – shelter with constrained access, this difference is possibly due to the fact that part of the task relies on the imprecise estimation of the distance provided by the range-and-bearing board.

AAC – aggregation with ambient cues. The results are reported in Figure 9. Both Chocolate and Vanilla perform significantly better than C-Human. The median recorder for Chocolate is slightly better than the one recorded for Vanilla. Concerning the performance difference between simulation and robot experiments, Vanilla shows a smaller difference with respect to Chocolate and C-Human.

5.2 Aggregate analysis

The results of the aggregate analysis are reported in Figure 10. The plot depicts, for each design method, the median rank over the five tasks, together with the corresponding 95% confidence intervals. If the intervals of two methods do not overlap, the difference between the corresponding median rank is statistically significant.

These results confirm those presented in Francesca et al (2014b): C-Human outperforms Vanilla. The better performance of C-Human over Vanilla corroborates
the hypothesis formulated in Section 1: as C-Human and Vanilla design control software combining the same modules, the failure of Vanilla to match C-Human’s performance is to be ascribed to Vanilla’s optimization algorithm. The hypothesis is confirmed by the fact that Chocolate, which adopts a more advanced optimization algorithm, outperforms Vanilla. This improvement of Chocolate is such that Chocolate outperforms also C-Human. In other words, under the experimental conditions defined by the protocol presented in Section 4, Chocolate produce better control software than the one produced by a human designer that operates on the same set of modules as Chocolate.

The results already presented in Francesca et al (2014b) show that C-Human outperforms U-Human, that is, the human designer that produces control software without any restriction on the structure of the control software. Together, the results presented in Francesca et al (2014b) and those presented here lead to a stronger statement: under the experimental conditions defined in Section 4, Chocolate designs control software that outperforms the one produced by a human designer, whether the human is constrained to use Chocolate’s (and Vanilla’s) modules or not.

6 Conclusions

In this paper we introduced Chocolate, an improved version of Vanilla. Chocolate generates control software by combining and fine-tuning a set of preexisting parametric modules. Chocolate and Vanilla operate on the same set of modules. The only difference between them is the optimization algorithm adopted to search the space of the possible designs: Chocolate adopts Iterated F-Race, an improved version of F-Race, which is the optimization algorithm adopted by Vanilla.

We conducted an empirical analysis on five tasks: shelter with constrained access, largest covering network, coverage with forbidden areas, surface and perimeter coverage, and aggregation with ambient cues. These tasks were defined by experts in the domain of swarm robotics that were unaware of the internals of Chocolate and Vanilla. The empirical analysis included Chocolate, Vanilla, and a manual design method, C-Human. C-Human is a design method in which a human combines manually the modules of Chocolate. In Francesca et al (2014b), C-Human is shown to perform better than Vanilla.

We conceived the analysis to corroborate three working hypothesis: (i) the weak point of Vanilla is its optimization algorithm; (ii) by adopting a more advanced optimization algorithm, Chocolate improves over Vanilla; and (iii) the improvement is such that Chocolate outperforms C-Human.

The results confirmed our working hypothesis: Chocolate performs better than C-Human. Chocolate is the first automatic design method that is shown to outperform a human designer.

Acknowledgements. We thank Maria Zampetti and Maxime Bussios for their help with the robots, and Marco Chiarandini for his implementation of the Friedman test. This research was partially supported by the COMEX project within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office. It received funding also from the European Research Council under the European Union’s Seventh Framework Programme—ERC grant agreement n. 246939 and
from the European Science Foundation via the H2SWARM project. Vito Trianni acknowledges support from the Italian CNR. Arne Brutschy, Franco Mascia, and Mauro Birattari acknowledge support from the Belgian F.R.S.-FNRS.

References


Birattari M (2009) Tuning Metaheuristics. Springer, Berlin, Germany


