Noname manuscript No.
(will be inserted by the editor)

Evolution of Hybrid Robotic Controllers for Complex Tasks

Miguel Duarte · Sancho Moura Oliveira · Anders Lyhne Christensen

the date of receipt and acceptance should be inserted later

Abstract We propose an approach to the synthesis of hierarchical control systems comprising both evolved and manually programmed control for autonomous robots. We recursively divide the goal task into subtasks until a solution can be evolved or until a solution can easily be programmed by hand. Hierarchical composition of behavior allows us to overcome the fundamental challenges that typically prevent evolutionary robotics from being applied to complex tasks: bootstrapping the evolutionary process, avoiding deception, and successfully transferring control evolved in simulation to real robotic hardware. We demonstrate the proposed approach by synthesizing control systems for two tasks whose complexity is beyond state of the art in evolutionary robotics. The first task is a rescue task in which all behaviors are evolved. The second task is a cleaning task in which evolved behaviors are combined with a manually programmed behavior that enables the robot to open doors in the environment. We demonstrate incremental transfer of evolved control from simulation to real robotic hardware, and we show how our approach allows for the reuse of behaviors in different tasks.

Keywords Evolutionary Robotics · Hierarchical Control · Artificial Evolution

This work was partly supported by FCT – Foundation of Science and Technology under grants SFRH/BD/76438/2011, PEst-OE/EEI/LA0008/2013 and EXPL/EEI-AUT/0329/2013.

Miguel Duarte (✉) · Sancho Moura Oliveira · Anders Lyhne Christensen
Instituto Universitário de Lisboa (ISCTE-IUL), Instituto de Telecomunicações, Avenida das Forças Armadas, 1649-026 Lisboa
E-mail:
{miguel_duarte,sancho.oliveira,anders.christensen}@iscte.pt

1 Introduction

Evolutionary robotics (ER) is a field in which evolutionary computation is applied to the design of autonomous robots. Artificial evolution is commonly employed to synthesize behavioral control [38], but can also be used to synthesize robot morphology (see [24, 6] for recent examples). ER techniques have the potential to automate the design of control systems without the need for manual and detailed specification of the desired behavior [14] and to exploit the way in which the world is perceived through the robot's (often limited) sensors. In ER, it is common to use artificial neural networks (ANNs) as robotic controllers because they: (i) provide evolution with a relatively smooth search space, (ii) are able to tolerate noisy input inherent to most real world sensors, and (iii) have been shown capable of representing general and adaptive solutions [15]. Numerous studies have demonstrated evolved control systems that enable robots to solve simple tasks in surprisingly elegant ways [38, 16, 25, 7, 36]. It has, however, proven difficult to evolve controllers for complex tasks [37].[1]

The evolutionary process follows a fitness gradient in the search space. If a task is too difficult, an initial randomly generated population may be located in a region of the search space without a fitness gradient. The evolutionary process may therefore drift in such a region and fail to *bootstrap*. Even when a gradient is present, the gradient may lead the evolutionary process toward low-quality local optima, a problem known as

[1] Task complexity cannot be quantified in general, and we therefore rely on the intuitive notion of complexity. We consider a task to be *complex*, if the task clearly is more challenging than the state of the art or if it incorporates several tasks that only individually have been solved by evolved controllers.

deception [48]. As the complexity of a task increases, the fitness landscape typically becomes rugged [37], and the evolutionary process becomes more vulnerable to deception.

A large number of candidate solutions typically have to be evaluated. As a consequence, evaluations are usually conducted in computer simulation and not on real robotic hardware. Despite best efforts to accurately simulate the real world, differences are bound to exist between simulation and reality. The differences between simulation and reality are often referred to as *the reality gap* [27]. The presence of the reality gap means that controllers evolved in simulation may exploit aspects of the simulated world that are different or may not exist in the real world. Controllers evolved in simulation are therefore not guaranteed to maintain their performance when executed on real robotic hardware [28].

The reality gap, the bootstrapping problem, and deception remain major challenges in ER. While there has been considerable progress in the field of ER in recent years, there have been no significant breakthroughs indicating that ER scales to tasks with the level of complexity found outside strictly controlled laboratory conditions. Notwithstanding, evolutionary techniques still have significant potential in controller design that can be realized if we depart from the tradition unadulterated application of ER and embrace more practical and engineering-oriented approaches.

In this paper, we propose an approach that combines the benefits of ER, such as automatic synthesis of control, with human engineering to circumvent the bootstrapping problem, to avoid deception, and to successfully cross the reality gap. If a robotic controller cannot be evolved for a particular task, we manually divide the task into two or more sub-tasks and evolve sub-controllers for each sub-task. An additional controller that selects which sub-controller is active at any given time is then evolved. Behavioral control for complex tasks can thus be synthesized in an incremental and hierarchical manner, and issues related to performance on real hardware can be addressed at each increment. Our approach allows for the reuse of evolved controllers and for the combination of evolved and preprogrammed control. Some tasks, such as those that require the robot to perform actions with a high degree of accuracy, might be difficult to simulate with sufficient fidelity to allow for successful transfer of evolved control to a real robot. For such tasks, a preprogrammed behavior can be developed directly for the real robotic hardware and integrated in the hierarchical structure of the controller. We demonstrate our approach in two distinct tasks, which are beyond the complexity level of current state of the art in ER. In both tasks, controllers evolved in simulation maintain their performance when transferred to a real robot.

The main contribution of this paper is the introduction of a hierarchical approach to the semi-automatic synthesis of behavioral control for real robotic hardware. Our approach solves three fundamental issues related to the use of ER techniques as an engineering tool: (i) incremental evolution: by taking an incremental, divide-and-conquer approach to evolution, bootstrapping issues and deception are avoided, (ii) scalability: as partial solutions are combined, fitness functions can be derived based on the immediate task decomposition, and an increase in fitness function complexity is thereby averted as increasingly complex tasks are considered, and (iii) transfer: sub-controllers can be tested incrementally on real robotic hardware and issues related to real robot performance can be addressed locally in the controller hierarchy. Our approach furthermore allows for seamlessly integration of preprogrammed control and evolved control.

The structure of this paper is as follows. In Section 2, we discuss the related work. In Section 3, we discuss how behavioral control can be synthesized hierarchically. In Section 4, we present the software and hardware platforms used in the experiments. Sections 5, 6 and 7 detail three different experiments that were conducted in order to assess the validity of the proposed methodology. Finally, we present the conclusions in Section 8.

2 Background and Related Work

ER emerged as a field in the beginning of the 1990's [40]. The use of artificial evolution frees the designer from manually specifying a robot's controller in detail and allows for the self-organization of behavior [38]. Soon after the emergence of ER as a research field, two main challenges became clear, namely, (i) that it often is non-trivial to ensure successful transfer of behavior evolved in simulation to real robots, and (ii) that the number of evaluations required meant that simulation had to be used extensively.

Several authors have proposed approaches to enable controllers evolved in simulation to maintain their performance on real robotic hardware. Three complementary approaches were proposed by Miglino et al. [33]: (i) the use of real robot sensor samples in simulation to narrow the reality gap, (ii) the introduction of a conservative form of noise in order for controllers to increase their tolerance to different sensory conditions, and (iii) the continued evolution of controllers for a few generations on a real robot. Continuing evolution on

real hardware allows controllers to adjust to minor differences between simulation and reality. The use of samples from real robots and a conservative form of noise has since become widespread in ER studies, but crossing the reality gap still proves a challenging task. In 1997, Jakobi [27] advocated the use of *minimal simulations* in which the simulator only implements the specific features of the real world that the experimenter deems necessary for a robot to complete its task. All other features are hidden in an envelope of noise to minimize the presence of simulation-only artifacts that can prevent successful transfer of evolved control to real robotic hardware. It is uncertain if Jakobi's minimal simulations are feasible to apply to more complex tasks, since they require that the experimenter is able to determine the set of relevant features and to build a *simple*, task-specific simulator based on this set.

Koos et al. [28] proposed *the transferability approach* that aims to ensure successful transfer of control evolved in simulation to real robotic hardware. The transferability approach is a multi-objective formulation of ER, in which controllers are evaluated not only on their simulated performance, but also on their real-robot performance. A surrogate model is created to avoid having to test every solution in real hardware, and can be updated with results from a periodical real-robot experiment or by interpolation. In this way, a solution that relies on certain robot-environment dynamics that are inaccurately implemented in simulation might not be selected for reproduction by evolution. Koos et al. successfully validated the approach in two robotic tasks: a navigation task with an e-puck robot and a gait task with an 8-DOF quadrupedal robot. The authors propose to evaluate the transferability of the best controller and update the surrogate model every five generations. Although frequent real robotic evaluations may be feasible for simple tasks for which a solution can be evolved in a few (100 or less) generations, it quickly becomes unfeasible if a solution for a more complex task is sought since several hundred or potentially thousands of generations might be required. Furthermore, some tasks might take too long to perform or it may be difficult to automatically measure performance of the real robot.

Numerous studies have demonstrated evolved controllers solving basic tasks in surprisingly simple and elegant ways. Nelson et al. [37] surveyed different types of fitness functions used in the field of evolutionary robotics. In the discussion of their findings, they state that evolutionary robotics may possibly "*generate autonomous systems with limited general abilities at some point in the future*". In their survey of more than one hundred different ER studies, there were only reports on the successful application of ER techniques to relatively simple tasks, such as locomotion and obstacle avoidance [16], goal homing [22], foraging [36], and phototaxis [47]. In a different survey, Meyer et al. [32] argued that "*the challenge is to move from basic robot behaviours to ever more complex, non-reactive ones*". The lack of successful applications to complex tasks can be attributed to the bootstrapping problem and deception.

An approach to avoiding deception was proposed by Celis et al. [5], which allows for non-expert users to interact with the evolutionary process by allowing them to guide evolution away from local optima. The approach was demonstrated on a simple navigation task with a deceptive fitness function, where the robot has to reach a goal by moving around an obstacle. The non-expert user can aid evolution by defining an intermediate way-point that the robot should pass through on its way to the target location.

Different incremental approaches have been studied as a means to overcome the bootstrapping problem and to enable the evolution of behaviors for complex tasks. In incremental evolution, the initial random population typically starts in a simple version of the environment. The complexity of the environment is then progressively increased as the quality of the evolved controllers improves [19, 7, 8]. An alternative approach is to decompose the task into different sub-tasks that are then learned sequentially [22, 10], or even to incrementally change the structure of the artificial neural network instead of modifying the environment or the task [42].

While a single ANN controller is sometimes trained in each sub-task sequentially, different modules can also be trained to solve different sub-tasks. Moioli et al. [34] used a homeostatic-inspired GasNet to control a robot in two sub-tasks: obstacle avoidance and phototaxis. The authors used two different sub-controllers that were inhibited or activated by the production and secretion of virtual hormones, and they were able to evolve a controller that activated the appropriate sub-controller depending on external stimulus and internal stimulus. Nolfi and Parisi [39] experimented with dividing a neural network into different modules in a garbage collection task. The robot had to grasp objects and release them outside of the environmental bounds. For their experiments, they used a Khepera robot with a gripper module. They divided the network's output layer into two modules that competed for activation. The controller evolved one of the modules to find and pick up the objects, and one to release them outside the bounds of the environment.

Stanley and Miikkulainen [42] introduced *Neuroevolution of Augmenting Topologies* (NEAT), an evolutionary algorithm that not only optimizes the weights

of the neural controller's connections, but also incrementally augments the controller's topology by adding new connections and neurons. NEAT has been shown to outperform traditional evolutionary techniques for some tasks and to produce solutions with minimal complexity. Recently, Lehman and Stanley introduced novelty search [31] as a means to avoid premature convergence and to overcome bootstrapping issues. In novelty search, behaviors are not scored based on a traditional fitness function, but based on behavioral novelty with respect to previously evaluated individuals. Since novelty search does not have a static objective, bootstrapping is typically not an issue, and the constant evolutionary pressure toward behavioral innovation means that novelty search is unaffected by deception. However, effective behavior characterizations are not always trivial do define [18].

In the past, several approaches have been proposed to the synthesis of controllers by hierarchical task decomposition. These studies have applied different techniques, such as genetic programming [30], neuroevolution [29, 1], or fuzzy logic control [46]. The tasks in which the approaches have been demonstrated are relatively simple, and have been shown to be solvable by traditional evolutionary techniques. The approach presented in this paper is distinct in a number of key aspects: (i) our approach allows for hybrid controllers in which preprogrammed control and evolved control are mixed. Hybrid controllers can take advantage of the benefits of ER, namely the automatic synthesis of behavior, and at the same time the use of manual programming for behaviors that would be infeasible to evolve; (ii) we test our approach in tasks that go beyond the state-of-the-art in terms of complexity. We further demonstrate transfer of behavioral control from simulation to real robotic hardware without loss of performance; and (iii) *derived fitness functions* are introduced as a solution to circumvent the otherwise increase in fitness function complexity as the tasks considered become more complex.

Preliminary studies of the experiments presented in Section 5 and Section 6 have previously been published in [11] and [12], respectively. In this paper, we present a methodology that combines our previous work, and we demonstrate the application of the methodology to a non-sequential task that requires accurate sensorimotor coordination.

3 Hierarchical Evolution of Robotic Controllers

In this section, we provide an introduction of how our hierarchical approach can be applied in general. This approach is demonstrated in specific tasks in Sections 5, 6 and 7.

3.1 Controller Structure

In our approach, we resort to manual division of the task into simpler sub-tasks when an evolutionary process is unable to find a solution to a task. Sub-controllers are then evolved to solve each sub-task, and the complete controller is composed of several sub-controllers arranged in a hierarchical structure (see Figure 1). Each node in the hierarchy is either a *behavior arbitrator* or a *behavior primitive* [30]. Behavior primitives are at the bottom of the controller hierarchy and directly control the actuators of the robot, such as wheels. If it is possible to find an *appropriate fitness function* for a given task, a behavior primitive composed of a single ANN is evolved to solve the task. An appropriate fitness function is one that (i) allows evolution to bootstrap, (ii) leads to controllers that are able to solve the task consistently and efficiently in simulation, and (iii) leads to controllers that maintain performance when transferred to real robotic hardware. In case an appropriate fitness function cannot be found, the task is manually and recursively divided into sub-tasks until an appropriate fitness functions has been found for each sub-task. We resort to manual programming of behavior primitives if an appropriate fitness function cannot be found for a sub-task that cannot be further divided. Previously evolved or preprogrammed behaviors can be reused if sub-tasks are sufficiently similar or if the behavior is general enough to be used in different tasks.

Sub-controllers are combined through the evolution or preprogramming of a behavior arbitrator. A behavior arbitrator receives either all or a subset of the robot's

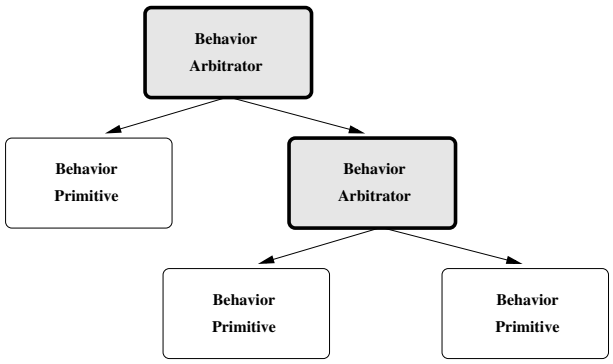

Fig. 1 A representation of the hierarchical controller. A behavior arbitrator delegates the control of the robot to one or more of its sub-controllers. A behavior primitive controls the actuators of the robots directly.

sensory inputs, and it is responsible for delegating control to one or more of its sub-controllers. The behavior arbitrators used in this study are all evolved and have one output neuron for each of their immediate sub-controllers. The output with the highest activation determines which sub-controller is activated at every control cycle. The state of a sub-controller is reset whenever it is deactivated. Alternative methods for activating sub-controllers could be used, such as allowing for multiple sub-controllers to be active at the same time.

The behavior primitives are synthesized first and are then combined through the synthesis of behavior arbitrators. The resulting controller can then be combined with other controllers to create a hierarchy of increasingly more complex behavioral control. Each time a new sub-controller (either a behavior primitive or a composed controller) has been synthesized, its performance on real robotic hardware can be evaluated. The experimenter can thus address issues related transferability incrementally during the development of the control system. A pseudocode representation of the proposed approach can be seen in Algorithm 1.

Algorithm 1 Pseudocode representation of the proposed approach.

```
procedure HIERARCHICALEVOLUTION(task)
    if suitable controller exists then
        return existing controller
    else if infeasible to simulate accurately then
        return PREPROGRAMCONTROLLER(task)
    else
        behaviorPrimitive = EVOLVEPRIMITIVE(task)
        if behaviorPrimitive performs adequately then
            return behaviorPrimitive
        else
            sub–tasks = DECOMPOSE(task)
            controllers = ∅
            for each sub–task do
                c = HIERARCHICALEVOLUTION(sub–task)
                add c to controllers
            end for
            return EVOLVEARBITRATOR(task,controllers)
        end if
    end if
end procedure
```

The logic in each node of the controller hierarchy is completely independent of the logic in other nodes. For the experiments presented in Sections 5, 6 and 7, for instance, we evolve continuous-time recurrent neural networks (CTRNN) [2] and use preprogrammed control, but nodes could be other types of ANNs or different types of control or decision systems altogether. Nodes can also be evolved under different simulation conditions and even using different algorithms. In this paper, we use a simple generational evolutionary algorithm and focus on issues related to the hierarchical composition of control. Our approach is, however, compatible with modern neuroevolution algorithms such as NEAT and novelty search, which can be used to evolve the individual components of our hierarchical controllers.

One of the potential costs of decomposing control hierarchically is that the solution space is restricted: the evolution of certain primitive behaviors is forced, and at the higher layers evolution is restricted by the set of behaviors available in lower layers. While decomposition might exclude optimal solutions, the narrower search space allows for the application of ER techniques to tasks that would otherwise be prohibitively complex.

3.2 Derived Fitness Functions

As we move up the controller hierarchy and attempt to synthesize behavioral control for increasingly complex tasks, fitness functions that lead to the evolution of adequate solutions may be challenging to define. If the fitness function for the evolution of a behavior arbitrator is difficult to define, it can be derived based on the task decomposition. The derived fitness function is constructed to reward the arbitrator for activating a sub-controller that is valid for the current sub-task, rather than for solving the global task. The use of derived fitness functions in the composition step circumvents the otherwise increase in fitness function complexity as the tasks considered become more complex. The experimenter may not always know which sub-controller is the optimal one for a given situational context, and for such cases, a subset of valid sub-controllers can be specified. In Section 5.2, we use a derived fitness function for the evolution of a behavior arbitrator.

4 Simulation, Evolution, and Robotic Hardware

We use the JBotEvolver simulation platform for evolution of behavioral control [9]. JBotEvolver is an open source, multirobot simulation platform, and neuroevolution framework. The simulator is written in Java and implements 2D differential drive kinematics.

In simulation, we run a total of ten evolutionary runs with a population size of 100 genomes for each node in a controller hierarchy. The fitness score assigned to each genome is the average score obtained in 50 simulations with different initial conditions. The five highest scoring genomes are copied directly to the next generation. Another 19 copies of each genome are made and mutation is applied to each gene with a probability of 10%. A Gaussian offset with a mean of 0 and

a standard deviation of 1 is applied when a gene undergoes mutation. The evolutionary runs for each node are terminated after an empirically determined number of generations. Upon termination, we conduct a post-evaluation of the highest scoring controller of each evolutionary run in a total of 100 samples for each different configuration of the environment. Genomes consist of floating-point alleles that encode the parameters of a CTRNN with one hidden layer of fully-connected neurons. The neurons in the hidden layer are governed by the following equation:

$$\tau_i \frac{dH_i}{dt} = -H_i + \sum_{j=1}^{J} \omega_{ji} I_j + \sum_{k=1}^{K} \omega_{ki} Z(H_k + \beta_k)$$

where τ_i is the decay constant, H_i is the neuron's state, J is the number of input neurons, ω_{ji} the strength of the synaptic connection from neuron j to neuron i, I is the set of input neurons, K is the number of output neurons, β is the bias term, and $Z(x) = (1 + e^{-x})^{-1}$ is the sigmoid function. The bias terms β_i, the decay constants τ_i, and the connection weights ω_{ji} are genetically controlled network parameters. The possible ranges of these parameters are: $\beta_i \in [-10, 10]$, $\tau_i \in [0.1, 32]$ and $\omega_{ji} \in [-10, 10]$. Circuits are integrated using the forward Euler method with an integration step-size of 0.2 and cell potentials are set to 0 when the network is initialized.

For our real hardware experiments, we used an e-puck [35] (see Figure 2). The e-puck is a small circular (diameter of 75 mm) differential drive mobile robotics platform designed for educational use. The e-puck's set of actuators is composed of two wheels that enable the robot to move at speeds of up to 13 cm/s, a loudspeaker, and a ring of eight LEDs. The e-puck is equipped with several sensors: (i) eight infrared proximity sensors which are able to detect nearby obstacles and changes in light conditions, (ii) three microphones (one near each wheel of the robot, and one toward the front), (iii) a color camera, and (iv) a 3D accelerometer. Additionally, our e-puck robots are equipped with a range & bearing board [21] which allows them to communicate with one another.

We use four of the e-puck's eight infrared proximity sensors: the two front sensors and the two lateral sensors. We collected samples (as advocated in [33]) from the sensors on a real e-puck robot in order to model them in JBotEvolver. Each sensor was sampled for 10 seconds (at a rate of 10 samples/second) at distances from an obstacle ranging from 0 cm to 12 cm. We collected samples at increments of 0.5 cm for distances between 0 cm and 2 cm, and at increments of 1 cm for distances between 2 cm and 12 cm.

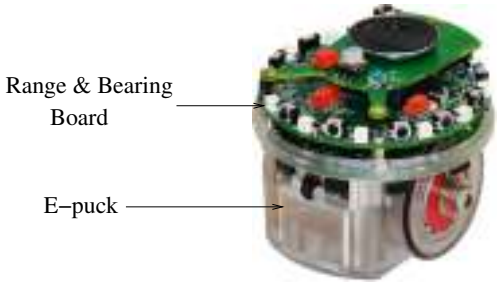

Fig. 2 The e-puck with a range & bearing board [21].

We use ray-casting to model the infrared sensors in the simulation. Seven rays are cast from each sensor in different directions, from $-\frac{\alpha}{2}$ to $\frac{\alpha}{2}$, where α is the sensor's opening angle. Based on experimental data from the robot, we used an α value of 70°. The distances at which the rays intersect with an obstacle are averaged and the final sensor reading is the interpolation of the closest samples collected on the real robot. Finally, noise is added to the reading with an amount based on the variance measured on the real robot for the particular distance.

The e-puck's infrared sensors can measure the level of ambient light. In the experiments presented in Section 5 and Section 6, we use ambient light readings from the two lateral proximity sensors to detect light flashes. When a light flash is detected on one of the sides, the activation of its corresponding sensor is set to 1. The sensor remains activated for 15 simulation cycles (equivalent to 1.5 seconds) to indicate that a flash has been detected. In simulation, we added Gaussian noise to the wheel speeds, with a standard deviation corresponding to 5% of the current wheel speed in each control cycle. The robot's speed is limited to 10 cm/s in our experiments.

The e-puck only has 8 kb of onboard memory. If the control code does not fit within the e-puck's limited memory, we run the control code off-board. When the control code is executed off-board, the e-puck starts each control cycle by transmitting its sensory readings to a workstation via Bluetooth. The workstation then executes the controller, and sends back the output of the controller (wheel speeds) to the robot. We use off-board execution of control code in the real robot experiments conducted in Section 5 and Section 7, and on-board execution of control code in the real robot experiments conducted in Section 6. Videos of our real-robot experiments and source code for our preprogrammed behaviors can be found online [13].

Our simulator allows to automate some parts of the hierarchical composition process. The experimenter defines a configuration file with the various sub-

controllers, including information on how they are connected and the evolutionary setup for each node (environment, controller, evolutionary algorithm, and evaluation function). The simulator then synthesizes and composes the hierarchical controller by picking the best controller for every hierarchical node, out of multiple runs. The experimenter can then test each node on the real robot. If a controller proves to be difficult to transfer to the real robot, the experimenter can change the configuration for that particular node and restart the evolutionary process from that node and up.

5 Evolving and Transferring Controllers for Complex Tasks

In this section, we apply the proposed methodology to a rescue task. The task is relatively complex and requires several behaviors typically associated with ER [37] such as exploration and obstacle avoidance [16], delayed response [45], and the capacity to navigate safely through corridors [41].

We purposefully designed the task to be more complex than any previously solved by real robots with evolved controllers: (i) a robot must first find its way out of a room with obstacles, (ii) the robot must then solve a double T-maze [3], and finally (iii) the robot must guide a teammate safely to the room. Variations of the T-maze task have been used extensively in studies of learning and motivation in animals [43], neuroscience [44], and robotics. In robotics, T-mazes have been used to study different neural network models such as diffusing gas networks [26], the online learning capability of continuous time recurrent neural networks [3], and the evolution of transferable controllers [27, 28]. While a T-maze task may appear simple from an anthropomorphic perspective, robots' sensors are often limited. In our experiment, each sensor only provides the controller with a single scalar or binary value. The information available to the controller about the environment and about the robot's position in the environment is thus very limited. Previous studies in which evolved control has been tested on real hardware, only single T-mazes were used.

A number of obstacles are located in the initial room. The room has a single exit that leads to the start of a double T-maze (see Figure 3). In order to find its teammate, the robot should exit the room and navigate to the correct branch of the maze. Two rows of flashing lights in the main corridor of the double T-maze give the robot information about the location of the teammate. Each row of lights indicates the branch leading to the teammate in a junction. For instance, if the left light of the first row and the right light of the second

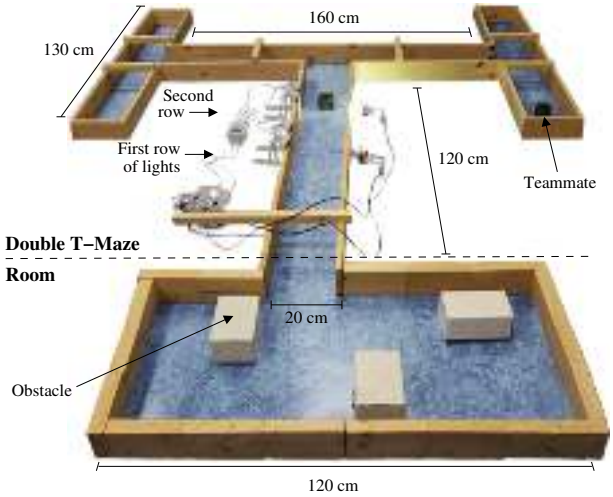

Fig. 3 The environment is composed of a room with obstacles and a double T-maze. The room is rectangular with varying side lengths. The double T-maze has a total size of 200 cm × 200 cm. The two rows with the lights are located in the central maze corridor. The activation of these two rows of lights indicates the location of a teammate.

row are activated, the robot should turn left at the first junction and right at the second junction. Upon navigating to the correct branch of the maze, the robot must guide the teammate back to the initial room. We included a boolean "near robot" sensor that indicates if the teammate is within 15 cm. For this sensor, we use readings from the range & bearing board.

For the real robot experiments, we built a double T-maze with a size of 200 cm × 200 cm (see Figure 3). In the real maze, a Lego Mindstorms NXT brick controlled the flashing lights. The brick was connected to four ultrasonic sensors that detected when the robot passed by. Lights were turned on by the first and third ultrasonic sensor and turned off by the second and fourth ultrasonic sensor. The NXT brick controlled the state of the lights using two motors.

We use simple, functional incremental fitness functions [37] for the evolution of behavioral primitives in the experiments presented below. Each fitness function typically has a number of cases that represent different outcomes of an experiment such as whether a robot reached its destination or not. Each case is kept simple and typically has only one or two terms. The cases are weighted by adding a constant and by multiplying by a factor, initially with values of zero and one, respectively. The values of the constants and factors are adjusted through an empirical trial-and-error process when necessary: if a bootstrapping case is exploited over a goal case, guiding evolution toward local optima, the relative weight of the exploited case is either decreased or the weight of the other cases are increased. While

the exact weights are not crucial, the relative weight between cases need to be such that solving the task is significantly better than not solving the task. For instance, bootstrapping cases should have significantly lower weights than high-level goal cases. The constants and factors that appear in the fitness functions below are the result of such a process. Given the simplicity of the fitness functions used, this process usually only required a couple of iterations.

5.1 Attempting to Evolve a Monolithic Controller

We attempted to evolve a monolithic controller, i.e. a controller with a single neural network for the complete rescue task. The chosen neural network was composed of seven input neurons (two light sensors, four infrared sensors, and one robot sensor), ten hidden neurons and two output neurons (two wheels). The robot was placed at a random position and with a random orientation in the room. The complexity of the task translates to a difficulty in finding an appropriate fitness function that allows evolution to bootstrap. In order to evaluate the controller, we chose a functional incremental, gradient-based fitness function with a bonus for reaching three intermediate points of the task: exiting the room, finding the teammate, and returning to the room. The fitness function is defined as follows:

$$f_{monolithic} = \sum_{i=1}^{3} \beta + \frac{D_i - d_s}{D_i}$$

where β is the bonus constant for reaching each intermediate point, D_i is total Euclidean distance from the previous intermediate point of task to the current intermediate point of the task, d_i is the robot's Euclidean distance to the next intermediate point of the task. The bonus constant of each term in the fitness function was chosen based on the methodology provided in Section 3.2, with values ranging from 1 to 5. The controllers were evolved for 1000 generations, and were post-evaluated in order to measure how many times they navigated to each of the intermediate points of the task. The controllers were able to solve the first part of the task and leave the room in 92% of the post-evaluation samples, and navigated to the correct exit of the double T-maze in 51% of the samples. The best controllers in some of the runs specialized in solving the task for a particular maze exit (i.e., 25% of the samples in which the robot managed to leave the room), and as a results, the average solve rate for the complete task was only 17%.

We also tried to evolve a monolithic controller for the rescue task using NEAT [42]. The NEAT implementation available in the open-source Encog Machine Learning Framework [23] was integrated in our simulator platform JBotEvolver [9]. The controllers were evaluated based on $f_{monolithic}$ in a total of 10 evolutionary runs. Each run was evolved for 1000 generations, and we used the parameter values proposed in [42]. We observed that evolution typically got stuck in a local optimum around the 300th generations. On average, the best controllers of the 10 evolutionary runs were able to leave the room in 77% of the samples, navigated to the correct exit of the room in 19% of the samples, but none of the controllers was able to return to the initial room. We experimented with several variations of the default parameter values, but obtained similar or inferior results (for details see [13]).

5.2 Decomposition of the Rescue Task

As shown in Section 5.1, the rescue task is relatively complex, which meant that we were unable to find an appropriate fitness function that allows evolution of a controller based on a single ANN. We therefore divided the rescue task into three sub-tasks: (i) exit the room, (ii) solve the double T-maze to find the teammate, and (iii) return to the room, guiding the teammate. We evolved sub-controllers to solve each of the sub-tasks. A hierarchical controller was then given access to each of the sub-controllers and evolved to solve the complete rescue task. Figure 4 shows the hierarchical structure of the complete controller, the description of each behavior arbitrator and behavior primitive, including the topology, inputs (sensors) and outputs (actuators) used by of each neural controller, and the performance results in simulation.

5.2.1 Exit Room Sub-task

The first part of the rescue task is an exploration and obstacle avoidance task. The robot is located in a room and must find a narrow exit leading to the maze. The room is rectangular with side lengths that vary between 100 cm and 120 cm drawn from a uniform distribution. We evolved the "Exit Room" behavior primitive to solve this sub-task. In each sample, we placed either two or three obstacles in the room depending on its size. Each obstacle was rectangular with random side lengths ranging from 5 cm to 20 cm drawn from a uniform distribution. The location of the room exit was also randomized in each trial. The robot was randomly oriented and positioned inside the room at the beginning of each

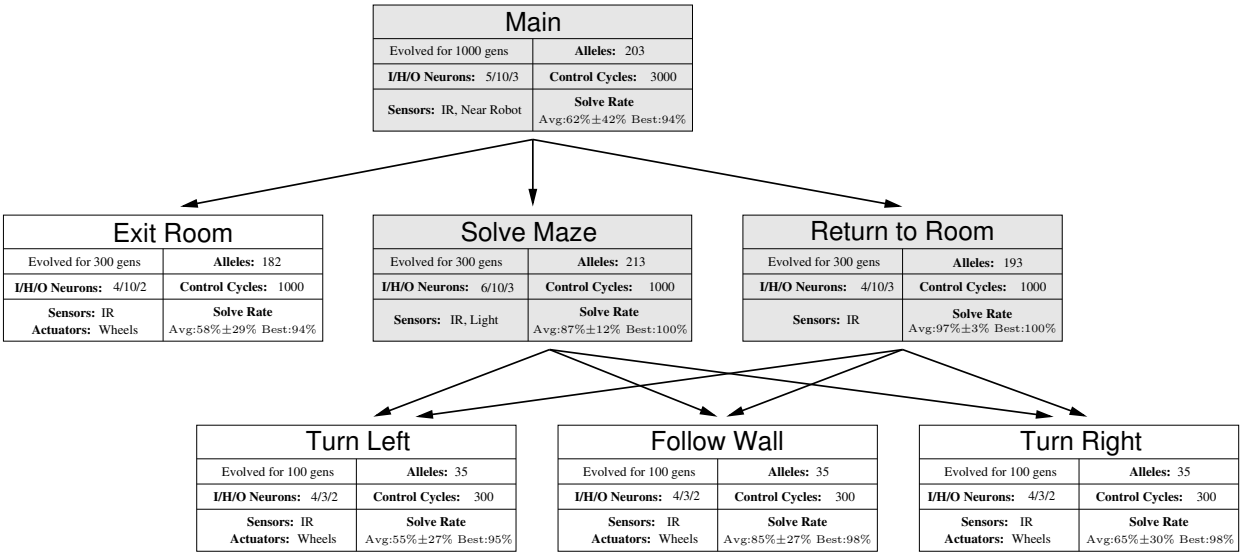

Fig. 4 Hierarchical Rescue Task Controller. The controller used in our experiments is composed of three behavior arbitrators (with darker background) and four behavior primitives. In each node, we list its name, the number of generations for which the sub-controller was evolved, the number of alleles, the number of input, hidden and output neurons, the sensors and actuators, the number of control cycles each evaluation lasted, and the average and best solve rate of the post-evaluation.

sample. Controllers were evaluated differently depending on whether they found the exit of the room within the allotted time or not, according to f_{exit_room}:

$$f_{exit\_room} = \begin{cases} 5 + \dfrac{C - c}{C} & \text{if exit was found} \\ \dfrac{D - d}{D} & \text{time expired} \end{cases}$$

where C is the maximum number of cycles (1 second = 10 control cycles), c is the number of cycles spent, D is the distance from the center of the room to its exit, and d is the closest point to the exit that the robot reached. f_{exit_room} rewards controllers that move from the center of the room and toward the exit. Since the controller has no information about the location of the exit, we reward it based on how close it got to the exit in order to implicitly promote exploration.

In two of the ten evolutionary runs, the highest scoring controller was able to find the exit of the room in over 90% of the post-evaluation samples. The best performing controller starts by moving away from the center of the room until it senses a wall, which it then follows counter-clockwise until the room exit is found. The remaining eight runs did not produce successful behaviors: the robots would spin around in circles, sometimes finding the exit by chance but often colliding with one of the walls or with an obstacle.

5.2.2 Solve Maze Sub-task

In the second sub-task, the robot has to solve a double T-maze in order to find a teammate that needs to be rescued (see Figure 3). Controllers were evaluated according to f_{maze}:

$$f_{maze} = \begin{cases} 1 + \dfrac{C - c}{C} & \text{if navigated to destination} \\ \dfrac{D - d}{3D} & \text{if collided or chose wrong path} \\ 0 & \text{if time expired} \end{cases}$$

where C is the maximum number of cycles, c is the number of cycles spent, D is the distance from the start of the maze to the robot's destination, and d is the final distance between the robot to its destination. f_{maze} awards a score of 1 plus a speed bonus in case the robot reaches the destination. If the robot collides with a wall or chooses a wrong path, a lower fitness is awarded, calculated based on how close to the destination it managed to get. The experiment ends prematurely if the robot either collides with a wall or reaches one of the maze exits. In such cases, we do not count the time as having expired.

We divided the "Solve Maze" sub-task into three different sub-tasks: "Follow Wall", "Turn Left" and "Turn Right", for which appropriate fitness functions could easily be specified. For each sub-task, we evolved a behavior primitive. Although the intended behaviors were

quite simple, we evolved the controllers in a wide variety of simple mazes, some of which had a high degree of difficulty due to the starting position and orientation of the robot. The difficulty of some of the mazes had an impact on the solve rates of these controllers, bringing the average down to 55% in the "Turn Left" controller and 65% in the "Turn Right" controller (see Figure 4). We used ten different mazes for the "Turn Left" and "Turn Right" controllers, and four mazes composed of corridors with different widths for the "Follow Wall" controllers.

We then evolved a behavior arbitrator with the three highest scoring behavior primitives as sub-controllers. At the beginning of each trial, the robot was placed at the start of the double T-maze and had to navigate to the correct branch based on the activations of the lights in the central corridor (see Figure 3). The robot was evaluated by f_{maze}, and the simulation sample was terminated if the robot collided with a wall or if it navigated to a wrong branch of the maze.

5.2.3 Return to Room Sub-task

The final sub-task consists of the robot guiding its teammate back to the room. We initially tried to evolve a behavior primitive for this sub-task, but the evolved solutions proved difficult to transfer successfully to the real robot. We therefore reused the behavior primitives previously evolved for maze navigation ("Follow Wall", "Turn Left" and "Turn Right"), and evolved a new behavior arbitrator. The new behavior arbitrator was evolved in the double T-maze with the robot starting in one of the four branches of the maze (chosen at random in the beginning of each trial). In the guidance sub-task, the robot had to navigate correctly through the maze, and we therefore used the same fitness function, f_{maze}, as in the "Solve Maze" sub-task. The only difference was the objective: the robot was evaluated based on how close it got to the initial room (see Figure 3), and not the distance to the teammate.

For the complete task, we evolved a behavior arbitrator with the highest scoring controllers for the "Exit Room", the "Solve Maze", and the "Return to Room" sub-tasks as sub-controllers. The teammate being rescued continuously emitted a signal while waiting for the rescuing robot. We used a derived fitness function, $f_{derived}$, to evolve the main behavior arbitrator for the rescue task. The derived fitness function rewards the selection of a valid behavior for the current sub-task and penalizes the selection of an invalid behavior (for instance, selecting the "Exit Room" behavior primitive if the robot was in the T-maze). The controller was awarded a fitness score between 0 and 1 for each sub-task (hence a maximum score of 3 for successful completion of all sub-tasks), depending on the proportion of the time that it selected the valid behavior, plus a time bonus. $f_{derived}$ is defined as follows:

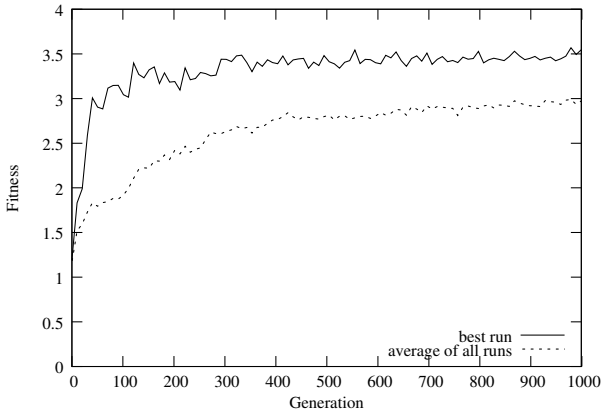

Fig. 5 The average fitness trajectory of each of the highest scoring controllers of all ten evolutionary runs, and the fitness trajectory of the highest scoring controller for the complete rescue task.

$$f_{derived} = \sum_{s=1}^{n} \left(\frac{t_s}{T_s} - \frac{w_s}{T_s} \right) + \begin{cases} \frac{C-c}{C} & \text{if the task completed} \\ 0 & \text{otherwise} \end{cases}$$

where the sum is over all the started sub-tasks, t_s is the number of cycles in which the controller chose a valid sub-controller for sub-task s, T_s is the number of simulation cycles that the controller has spent in sub-task s, w_s is the number of cycles in which the controller chose an invalid sub-controller for sub-task s, C is the maximum number of cycles, and c is the number of cycles spent. The fitness trajectory for the highest scoring controller evolved and the average fitness trajectory of all ten evolutionary runs for the rescue task can be seen in Figure 5. The best controller had a solve rate of 94%. Out of the ten controllers, seven were able to consistently solve the whole rescue task in over 89% of the samples, while the remaining three were not able to solve the three sub-tasks.

5.3 Transfer to the Real Robot

We tested each sub-controller incrementally on the real robot, which enabled us to ensure the transferability of the complete controller. After evaluating all the different evolutionary runs of the complete controller, we tested the best performing controller from the simulation on a real e-puck. The robot had to solve the complete rescue task: find the exit of the initial room, navi-

gate to the correct branch of the double T-maze, and return to the room. We used a room with a size of 120 cm × 60 cm for our real robot experiments. Three identical obstacles with side lengths of 17.5 cm and 11 cm were placed in the room as shown in Figure 3. We sampled the controllers six times for each light combination, resulting in a total of 24 samples.

To avoid interference between the two robots, we excluded the teammate in the real robot experiments, and manually triggered the near-robot sensor when the robot reached the correct maze branch. We ran additional proof-of-concept experiments in which we included a teammate that was preprogrammed to follow the main robot back to the initial room. Videos of these experiments can be found online [13].

The controller solved the composed task on the real robot in 22 out of 24 samples (a solve rate of 92%). The main behavior arbitrator consistently chose the correct sub-controller at each point of the task, and only failed once in the "Solve Maze" behavior (the robot turned the wrong way in the second intersection of the maze), and once in the "Return to Room" behavior (the robot did not turn at the intersection, ending up in a different maze branch).

The experiments and results presented above demonstrate how controllers can be composed in a hierarchical fashion to allow for the evolution of behavioral control for a complex task. For the main behavior arbitrator, we used a fitness function directly derived from the immediate decomposition of the task, that is, we used a fitness function that rewards a controller for activating a valid sub-controller given the current situational context. During evolution, an arbitrator (an ANN) was rewarded for (i) activating the "Exit Room" sub-controller while the robot was in the room, (ii) the "Solve Maze" sub-controller while the robot was in the maze, and (iii) the "Return to Room" sub-controller after the teammate had been located. In this way, we avoid that the complexity of the fitness function increases with the task complexity as sub-controllers are combined. The highest scoring controller in simulation was able to cross the reality gap, achieving a performance on real robotic hardware similar to the performance obtained in simulation.

6 Hybrid Controllers

In this section, we experiment with the evolution of hybrid control systems that can take advantage of preprogrammed behavior primitives. There are examples in literature where researchers have combined evolved control and preprogrammed control, but it has been done in an ad-hoc manner (see for instance [20]). Our approach, on the other hand, allows for a structured integration of learned and preprogrammed behavior in a hierarchical and incremental manner. We used the double T-maze task (see Section 5) with simple preprogrammed behavior primitives ("Follow Wall", "Turn Left", and "Turn Right"). The controller for the complete task is composed of an evolved behavior arbitrator that activates one of the three preprogrammed behavior primitives.

Two preprogrammed primitives take more than one control cycle to complete, namely turning 90° left or right. For such primitives, we introduce the concept of *locking* the controller. While a locking preprogrammed behavior is executing, no other behavior primitive can be executed. The locking mechanism helps to guarantee transfer of control from simulation to a real robot by ensuring that preprogrammed behaviors are allowed to complete before another behavior is executed.

6.1 Experiments and Results

The input layer of the ANN-based behavior arbitrator was composed of six neurons: one for each of the four infrared proximity sensors, and one for each of the two light sensors. The highest value of the output neurons of the neural network determines which one of the three possible preprogrammed behaviors is activated: "Follow Wall", "Turn Left" or "Turn Right". The complete controller and the results are shown in Figure 6.

After the evolutionary process was concluded, we conducted a post-evaluation of the evolved controllers in which the fitness of every controller was sampled 100 times for each of the four possible light configurations. The evolved controllers had an average solve rate of 50%(±31%). A solve rate of 80% or more was achieved by three of the ten controllers, with a solve rate of 93% for the highest scoring controller. The solutions produced in different evolutionary runs were similar. The controllers learned to navigate the T-maze correctly, but some were not able to take advantage of the information from the light flashes to consistently make the correct decisions at the junctions, which caused them to navigate to a wrong maze branch.

The highest scoring controller was tested on a real e-puck 24 times, six for each light configuration. The controller was successful in 22 of the 24 samples (a solve rate of 92%). In both failed samples, the robot turned to the wrong maze branch in the second intersection. These results are comparable to the results obtained in simulation and to the real robot experiments where the behavior arbitrator had access to evolved behavior primitives (see Section 5).

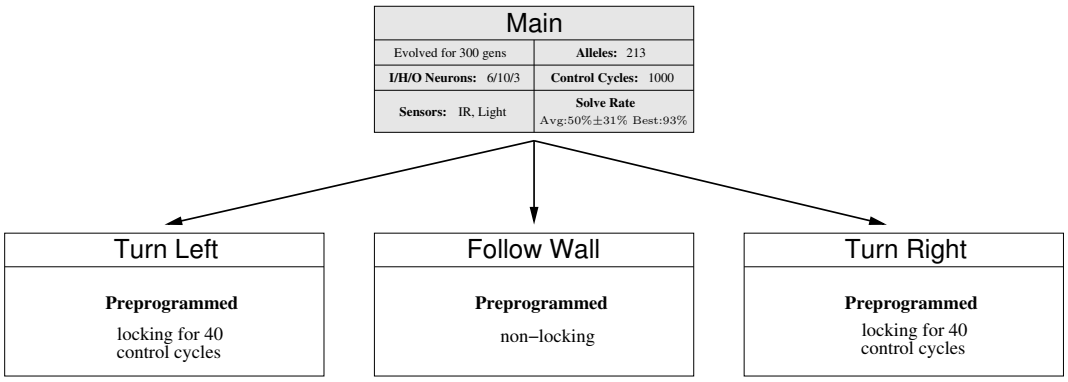

Fig. 6 Hierarchical Solve T-Maze Controller. The controller used in our experiments is composed of one behavior arbitrator and three preprogrammed behavior primitives. Both the "Turn Left" and the "Turn Right" preprogrammed behavior primitives lock the network during execution, in order to ensure that the behavior completes before another primitive can be executed.

We used the same preprogrammed behaviors both in simulation and on the real robot for the experiments presented in this section. Preprogrammed behavior primitives can be an alternative to evolved behavior primitives. It may not always be possible to evolve a controller for a sub-task, either because an appropriate fitness function cannot be found or because *fine sensorimotor behaviors* are required. Fine sensorimotor behaviors are those that require the robot to perform actions with a high degree of accuracy, and that depend on the precise information provided by the robot's sensors. When such behaviors are necessary for solving the task, evolved control may be combined with preprogrammed behaviors, which can be fine-tuned manually for the real robotic hardware. The use of preprogrammed behaviors is particularly beneficial for sub-tasks and behaviors that are infeasible to simulate with sufficient accuracy to allow for the successful transfer to real robotic hardware. Behaviors that have only been implemented for the real robotic hardware can be integrated in simulation in two different ways: samples can be collected from the real robot performing the preprogrammed behavior and then played back in simulation (*sensor playback*), or the normal control cycle can be stopped completely while the robot executes the preprogrammed behavior (*offline behavior*). In either case, it is only necessary to change the state of the environment according to the robot's ations, and not to simulate the detailed robot-environment interaction. We explore the *offline behavior* technique in the experiments described in the next section. Source code for the preprogrammed behaviors can be found online [13].

7 Hierarchical Evolution for Integrated Tasks

In the experiments presented in the previous sections, the behavior primitives were either all evolved, or all manually programmed. Furthermore, in the case of the experiments in Section 5, the task was sequential, which meant that the robot had to learn a strict sequence of behaviors: "Exit Room", "Solve Maze" and "Return to Room". In this section, we evaluate our approach in a task that is non-sequential and we combine evolved behavior primitives and preprogrammed primitives. We use the *offline behavior* technique described in Section 6, in which a preprogrammed behavior is implemented for the real robot, but not in simulation. When the controller activates such a preprogrammed behavior in simulation, the normal control cycle is stopped, the end-result of the robot-environment interaction is applied, and controller is then resumed.

We use a task in which the robot must clean dust spots. The dust spots appear in two rooms that are connected by a corridor (see Figure 7). The rooms are square-shaped and side lengths vary between 80 cm and 120 cm drawn from a uniform distribution. A new dust spot is randomly placed in one of the two rooms every 10 s. The placement of a new spot is determined by a Bernoulli trial with a probability p that the spot is placed in one room and $1-p$ that it is placed in the other room. The probability p is randomly sampled from the uniform distribution at the beginning of each trial. A maximum of five dust spots can be in the environment at any given time. This means that if the robot keeps cleaning one room for a long time, all dust spots may eventually be in the other room.

In order to traverse the corridor connecting the two rooms, the robot must first push a button to open both

doors (see Figure 7). We preprogrammed a behavior primitive to enable an e-puck to push a button, which opens the doors to the corridor. Pushing a button to open the doors requires fine sensorimotor coordination, since the buttons are difficult to detect and hit. The buttons are only 2.5 cm in diameter, and they must be pushed at an angle under 45°. This is a difficult interaction to model correctly in simulation, and it can also be a challenging behavior to evolve and transfer successfully. The preprogrammed "Push Button" behavior was therefore only implemented for the real robot, and activating the preprogrammed behavior automatically opens the door instantly in simulation. On the real robot, the preprogrammed behavior uses the e-puck's on-board camera to find and move the robot toward the button. When the preprogrammed behavior primitive that opens the door is activated, the control cycle of the main behavior arbitrator is stopped. The preprogrammed behavior primitive rotates the robot up to a maximum of 360° in order to try to locate the button, which can be identified by its red color. The central horizontal line of each image captured by the e-puck's camera is scanned. If the button is identified, the robot aligns itself, moves forward, and pushes it. The main behavior arbitrator is resumed after the preprogrammed behavior terminates. When the preprogrammed behavior is activated in simulation, the elapsed time of the current sample is increased by the average amount of time taken by the preprogrammed "Push Button" behavior on the real robot. Source code for the preprogrammed "Push Button" can be downloaded from [13].

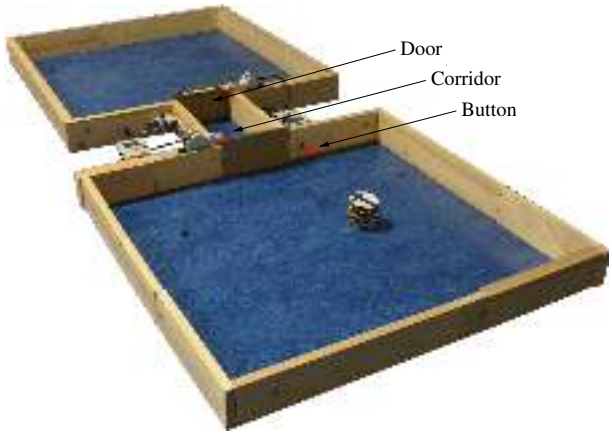

Fig. 7 The environment is composed of two rooms, connected by a corridor. The corridor is blocked by two doors that the robot can open by pushing a red button.

7.1 Experiments and Results

In the real robot experiments, virtual dust spots are implemented using a visual tracking system. The position and orientation of the robot is tracked using an overhead camera. The robot has a cross-shaped marker on top (see Figure 7), which is extracted from the video feed using the OpenCV library [4]. This information allows virtual dust spots to be detected by the robot using virtual sensors. The controller is executed in a workstation, and the resulting movement commands are sent to the robot every 100 ms. The robot is able to sense the dust spots up to 30 cm away with eight virtual sensors, positioned on the perimeter of the robot's body at equal angular intervals. The robot must activate a virtual actuator when it is within 5 cm of a dust spot to remove the spot. The robot is also equipped with two button sensors that are placed at angles of $-30°$ and $30°$, providing the controller with information on the direction and distance (up to 1 m) to the nearest button. The controller we used for these experiments can be seen in Figure 8. We decomposed the task into two main sub-controllers: "Change Room" and "Clean". The "Change Room" sub-controller is a behavior arbitrator that can choose between the "Open Door" behavior arbitrator and the "Enter Corridor" behavior primitive.

The robot must push a button to open the doors to the corridor before it can move from one room to the other. Once the doors have been opened, the robot has 40 seconds to traverse the corridor that leads to the other room before the doors close. We divided the "Change Room" sub-controller into a behavior primitive and a behavior arbitrator: the "Open Door" behavior arbitrator, which moves the robot toward the current room's button ("Move to Button" behavior primitive) and pushes the button ("Push Button" preprogrammed behavior primitive), and the "Enter Corridor" behavior primitive, which navigates to the corridor and crosses to the other room.

We first evolved the "Move to Button" behavior primitive. The controller was evaluated by by f_{two_rooms} according to a distance gradient to the button, with a time-dependent bonus upon correctly activating the preprogrammed behavior primitive:

$$f_{two\_rooms} = \begin{cases} 1 + 10 \cdot \dfrac{C - c}{C} & \text{if achieved objective} \\ \dfrac{D - d}{D} & \text{otherwise} \end{cases}$$

where C is the maximum number of cycles, c is the number of cycles spent, D is the distance from the robot's

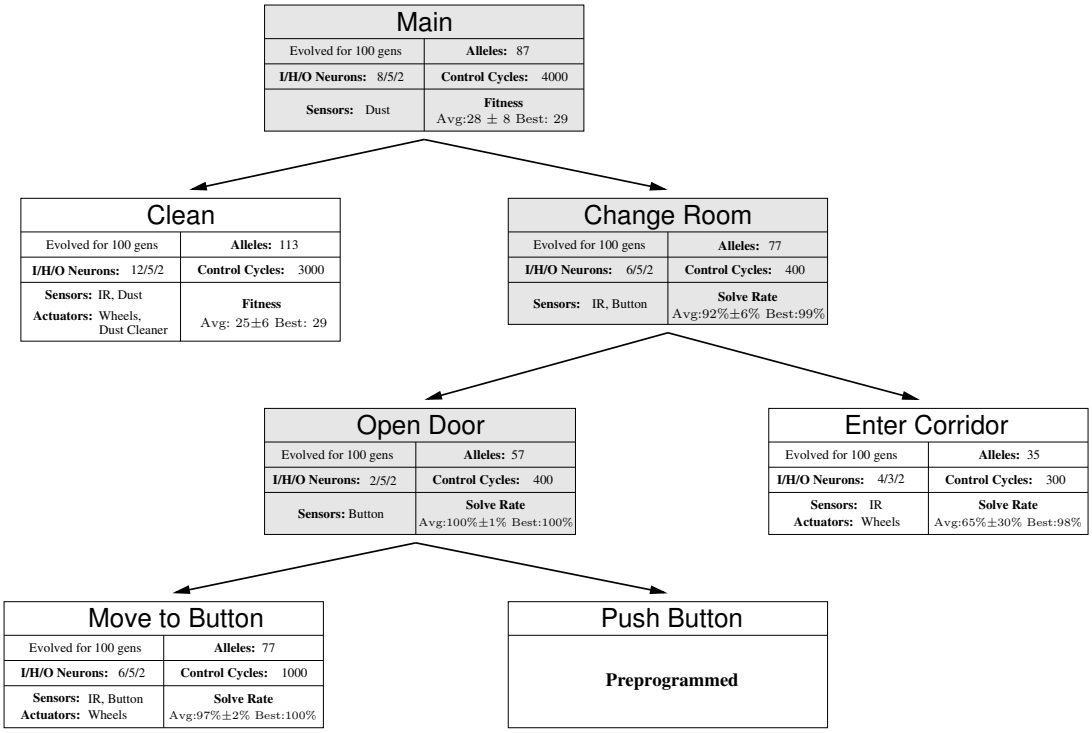

Fig. 8 Hierarchical Dust Cleaning Controller. The controller synthesized for the dust cleaning experiment is composed of three behavior arbitrators and four behavior primitives.

starting point to the button, and d is the closest point to the button that the robot reached. In the case of the "Move to Button" behavior primitive, the objective was to get within 5 cm of the button.

The "Open Door" behavior arbitrator has access to the button sensor. In simulation, the preprogrammed "Push Button" behavior opens the doors if the robot is within 20 cm of the button, otherwise the robot stops moving while the behavior is activated. Since the "Push Button" behavior takes some time to execute on the real hardware, controllers that activate this behavior too often or too far from the button are therefore indirectly penalized because they have less time to clean. On the real robot, such controllers would make the robot search for the button at a distance that would make it very difficult for the e-puck's camera to detect a small target. The controllers were evaluated by f_{two_rooms}, where the objective was to open the corridor doors. Since the button was placed to the right-hand side of each door, we reused the "Turn Right" behavior primitive from Section 5 instead of evolving a new primitive for the "Enter Corridor" sub-controller. This behavior primitive allows the robot to move to the other room after pushing the button.

For the evolution of the "Change Room" behavior arbitrator, the robot was positioned and oriented randomly in one of the rooms at the beginning of each trial, and had to push the button, enter the corridor and move to the other room. Each controller was also evaluated according to f_{two_rooms}, where the objective was to reach the other room. For the "Change Room" arbitrator, D is the distance from the robot's starting point to the end of the corridor, and d is the point closest to the end of the corridor that the robot reached.

The "Clean" behavior primitive was evolved in a single room without any door or exit. The output neurons controlled the speed of the robot's wheels, and the "clean dust spot" actuator was triggered if its corresponding output had an activation value higher than 0.5. If no dust spot is nearby when the robot activates the "clean dust spot" actuator, the speed of the robot is set to 0. This penalizes the controller from always activating the actuator, instead of only activating it near a dust spot. The robot was randomly oriented and positioned near the center of the room at the beginning of each sample. The fitness function rewarded the robot based on how many dust spots it cleaned (a score of 1 for each dust spot) in the allotted time or until it collided with a wall.

To obtain a controller for the complete task, the "Clean" behavior primitive and the "Cross Rooms" behavior arbitrator were combined via the evolution of a new behavior arbitrator (see Figure 8). All evolutionary runs converged to a behavior where the robot

would move between rooms whenever necessary. If the robot did not sense any dust spot for a certain amount of time, the "Main" arbitrator would choose to activate the "Change Room" behavior arbitrator. The controllers were evaluated according to the number of dust spots they cleaned, as in the evolution of the "Clean" behavior primitive. The controllers achieved an average fitness of 28 ± 8 in the ten evolutionary runs, and the best controller achieved an average of 29 ± 6. The controllers chose to cross to the other room an average of 2.86 times per sample.

7.2 Real Robot Experiments

For our real robot experiments, we built the walls of the two rooms and the corridor using wooden blocks. Each room had a size of 100 cm × 100 cm, and the corridor had a length of 40 cm and a width of 18 cm (see Figure 7). Both doors were opened and closed by motors connected to a Lego Mindstorms NXT brick. A physical button was placed on the wall to the right of the entrance to the corridor in each room.

We transferred the highest scoring controller from simulation to the real e-puck. The performance of the controller on the real robot was sampled five times in five different configurations, for a total of 25 samples. We used fixed rates at which the dust spots were placed in each room. In each configuration, the rate of a dust spot being placed in each room was set to one of the following: 1 : 0, 3 : 1, 1 : 1, 1 : 3, and 0 : 1. The robot started each sample in one of the rooms. In the 3 : 1 scenario, for instance, three dust spots are placed in the room where the robot starts, and then one dust spot is placed in the other room, and so on. The dust spots were placed deterministically at these fixed rates in order to allow for direct comparison of performance in simulation and performance in real hardware. We sampled the robot's performance in simulation 100 times per configuration. Each sample lasted five minutes (3000 control cycles). The results can be seen in Figure 9. In the real robot experiments, the controller was able to complete the task with a performance comparable to that obtained in simulation.

Several outliers can be observed in the 0 : 1 configuration. In this configuration, the robot starts in a room where no dust spots are ever placed. After some time, the controller decides to cross to the other room. In most of the samples, the robot keeps cleaning the room in which all the dust spots appear, but sometimes the controller might choose to return to the initial room. This happens when several dust spots are placed in a cluster and the robot does not explore the part of the room containing the cluster for a certain amount of time. In such cases, the number of dust spots that the robot can clean are significantly lower than if the robot remains in the room in which dust spots appear. The robot wrongly returned to the initial room in one of the real robot samples. As a result the robot only cleaned 13 dust spots in the 0 : 1 experimental setup, and received a low fitness (see outlier for the 0 : 1 setup in Figure 9).

7.3 Discussion

In this section, we introduced a non-sequential task that required fine sensorimotor coordination. A hybrid approach in which evolved behaviors are combined with preprogrammed behaviors can be beneficial when some behaviors are too difficult to simulate accurately. In the case of pushing the button, we did not implement the preprogrammed behavior in simulation. When the controllers were being evolved, the doors in the environment would open automatically if the robot activated the "Push Button" behavior primitive near the button. On the real robot, activating that same behavior primitive triggered a preprogrammed behavior, which used the robot's camera to detect and push a physical button in the environment.

In the experiment, we reused a previously evolved behavior primitive from a different experiment. The "Turn Right" behavior primitive from Section 5 was used to navigate from one room to the other. In this case, the behavior could be reused because of the characteristics of both the task and the environment: the button was positioned to the right of the door, and the robot had to turn right to enter the corridor af-

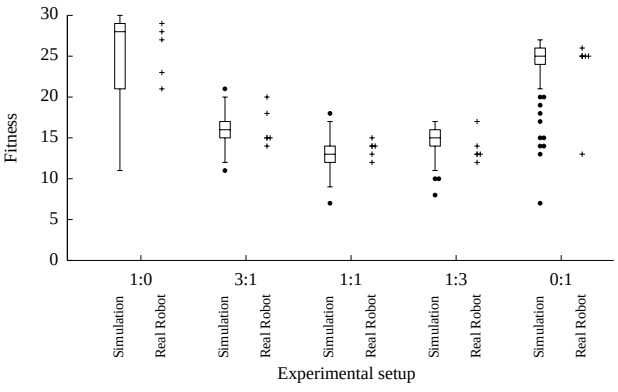

Fig. 9 Results of the real robot experiments in the dust cleaning experimental setup. The box plots represent 100 samples in simulation, while the scatter plots represent the fitness obtained in the real robot experiments. The whiskers extend to the most extreme data point within 1.5× the interquartile range.

ter pushing the button. If the position of the button was unknown, it would have been necessary to evolve a new behavior for the sub-task. Classic evolutionary approaches force the designer to evolve new controllers for similar tasks. Our approach, on the other hand, allows for the reuse of previously evolved or preprogrammed behaviors.

8 Conclusions

The application of classic evolutionary robotics techniques to the synthesis of controllers for complex tasks has proven problematic: the evolutionary process is often challenging to bootstrap and vulnerable to deception when the task is difficult. Successful transfer of evolved control from simulation to real hardware also remains a largely unsolved problem. In this paper, we demonstrated how hierarchical composition of robotic controllers can overcomes these issues. We recursively divide the goal task into sub-tasks until an appropriate fitness function has been found or until a behavior can easily be programmed by hand. In this way, partial solutions can be found incrementally and successful transfer to real robotic hardware can be guaranteed at each increment. Controllers for complex tasks can thus be synthesized in a hierarchical fashion while, at the same time, they can benefit from evolutionary robotics techniques, namely (i) automatically synthesis of control, and (ii) evolution's ability to exploit the way in which the world is perceived through the robot's (often limited) sensors. In summary, our contribution is threefold: (i) we combined preprogrammed and evolved control, (ii) we demonstrated our approach on real robotic hardware and in tasks that are beyond the state of the art in terms of task complexity, and (iii) we introduced the concept of derived fitness functions that circumvent an increase in fitness function complexity as progressively more complex tasks are considered.

Some researchers advocate the use of implicit, behavioral, and internal fitness functions [17]. Fitness functions with such characteristics, in theory, allow for solutions to emerge through an autonomous self-organization process. In practice, however, such fitness functions, which are supposed to be redeemed from any constraints imposed by a priori knowledge, are often the result of a series of unsuccessful experiments. The fitness function is modified after each unsuccessful experiment based on the results and on the experiment's guess concerning what may be "wrong". While we do not dismiss the potential benefits of implicit, behavioral, and internal fitness functions, we instead suggest dividing the task into sub-tasks, when such a function cannot easily be found.

The transfer of behavioral control from simulation to a real robot is usually a hit or miss because a controller for the goal task is completely evolved in simulation before any tests are conducted on real hardware. In our approach, the transfer from simulation to real robotic hardware can be conducted in an incremental manner as sub-controllers are evolved. Crossing the reality gap one sub-controller at a time allows the designer to address issues related to transferability immediately and locally in the controller hierarchy. The fact that each sub-controller solves only part of the task also allows for the use of different types of noise and other sub-task specific configurations in simulation.

The potential cost of an engineered approach, such as the approach proposed in this paper, is that evolution is constrained. The solution space for a behavior arbitrator is determined by the set of behavior primitives available, and the space is therefore smaller than the solution space in a traditional ER setup, in which the neural controller has direct control over the robot's actuators. The restricted solution space may exclude the optimal solution(s) for a given robot and task and surprisingly simple and elegant solutions that the experimenter did not foresee may therefore never be discovered. The solution space, however, is only restricted when evolution is not able to find a solution for the task, and the combination of evolutionary computation, task decomposition, and preprogramming enables the application of ER to tasks that would otherwise be insuperable.

References

1. J. A. Becerra, F. Bellas, J. S. Reyes, and R. J. Duro. Complex behaviours through modulation in autonomous robot control. In *Proceedings of the 8th International Work-Conference on Artificial Neural Networks (IWANN)*, volume 3512 of *Lecture Notes in Computer Science*, pages 717–724. Springer, Berlin, Germany, 2005.
2. R. D. Beer and J. C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992.
3. J. Blynel and D. Floreano. Exploring the T-Maze: Evolving learning-like robot behaviors using CTRNNs. In *Applications of Evolutionary Computing*, pages 593–604. Springer, Berlin, Germany, 2003.
4. G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 25(11), 2000.
5. S. Celis, G. S Hornby, and J. Bongard. Avoiding local optima with user demonstrations and low-level control. In *Proceedings of the IEEE Congress*

on Evolutionary Computation (CEC), pages 3403–3410. IEEE Press, Piscataway, NJ, 2013.

6. N. Cheney, R. MacCurdy, J. Clune, and H. Lipson. Unshackling evolution: Evolving soft robots with multiple materials and a powerful generative encoding. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. ACM Press, New York, NY, 2013. in press.
7. A. L. Christensen and M. Dorigo. Evolving an integrated phototaxis and hole avoidance behavior for a swarm-bot. In *Proceedings of 10th International Conference on the Simulation and Synthesis of Living Systems (ALIFE)*, pages 248–254. MIT Press, Cambridge, MA.
8. A. L. Christensen and M. Dorigo. Incremental evolution of robot controllers for a highly integrated task. In *From Animals to Animats 9: Proceedings of The 9th International Conference on the Simulation of Adaptive Behavior (SAB)*, pages 473–484. Springer, Berlin, Germany, 2006.
9. A. L. Christensen, S. Oliveira, and M. Duarte. JBotEvolver, 2014. URL `http://code.google.com/p/jbotevolver`.
10. R. de Nardi, J. Togelius, O. E. Holland, and S. M. Lucas. Evolution of neural networks for helicopter control: Why modularity matters. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, pages 1799–1806. IEEE Press, Piscataway, NJ, 2006.
11. M. Duarte, S. Oliveira, and A. L. Christensen. Hierarchical evolution of robotic controllers for complex tasks. In *IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–6. IEEE Press, Piscataway, NJ, 2012.
12. M. Duarte, S. Oliveira, and A. L. Christensen. Automatic synthesis of controllers for real robots based on preprogrammed behaviors. In *From Animals to Animats 12: Proceedings of the 12th International Conference on Adaptive Behaviour (SAB)*, pages 249–258. Springer, Berlin, Germany, 2012.
13. M. Duarte, S. Oliveira, and A. L. Christensen. Videos and source code from real robot experiments, 2014. URL `http://home.iscte-iul.pt/~alcen/hybridcontrollers/`.
14. D. Floreano and L. Keller. Evolution of adaptive behaviour in robots by means of Darwinian selection. *PLoS Biology*, 8(1):1–8, 2010.
15. D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats 3: Proceedings of the 3rd International Conference on Simulation of adaptive behavior (SAB)*, pages 421–430, MIT Press, Cambridge, MA, USA, 1994.
16. D. Floreano and F. Mondada. Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics–Part B: Cybernetics*, 26(3):396–407, 1996.
17. D. Floreano and J. Urzelai. Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13(4–5):431–443, 2000.
18. J. Gomes and A. L. Christensen. Generic behaviour similarity measures for evolutionary swarm robotics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation Conference (GECCO)*, pages 199–206, ACM, New York, NY, USA, 2013.
19. F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 3-4(5):317–342, 1997.
20. R. Groß, M. Bonani, F. Mondada, and M. Dorigo. Autonomous self-assembly in swarmbots. *IEEE Transactions on Robotics*, 22(6):1115–1130, 2006.
21. A. Gutiérrez, A. Campo, M. Dorigo, D. Amor, L. Magdalena, and F. Monasterio-Huelin. An open localization and local communication embodied sensor. *Sensors*, 8(11):7545–7563, 2008. ISSN 1424-8220.
22. I. Harvey, P. Husbands, and D. Cliff. Seeing the light: artificial evolution, real vision. In *From Animals to Animats 3: Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior (SAB)*, pages 392–401. MIT Press, Cambridge, MA, 1994.
23. J. Heaton. Encog machine learning framework, 2008. URL `http://www.heatonresearch.com/encog`.
24. J. Hiller and H. Lipson. Automatic design and manufacture of soft robots. *IEEE Transactions on Robotics*, 28(2):457–466, 2012. ISSN 1552-3098.
25. G. S. Hornby, S. Takamura, T. Yamamoto, and M. Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.
26. Phil Husbands. Evolving robot behaviours with diffusing gas networks. In *Proceedigs of the 1st European Workshop Evolutionary Robotics (EvoRobot)*, pages 71–86. Springer, Berlin, Germany, 1998.
27. N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6 (2):325–368, 1997.
28. S. Koos, J.-B. Mouret, and S. Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, 17(1):122–145, 2013.

29. T. Larsen and S.T. Hansen. Evolving composite robot behaviour - a modular architecture. In *Proceedings of the 5th International Workshop on Robot Motion and Control (RoMoCo)*, pages 271–276. IEEE Press, Piscataway, NJ, 2005.
30. W.-P. Lee. Evolving complex robot behaviors. *Information Sciences*, 121(1-2):1–25, 1999.
31. J. Lehman and K.O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.
32. J.-A. Meyer, P. Husbands, and I. Harvey. Evolutionary robotics: A survey of applications and problems. In *Proceedings of the 1st European Workshop on Evolutionary Robotics (EvoRobot)*, pages 1–21. Springer, Berlin, Germany, 1998.
33. O. Miglino, H. H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1996.
34. R. C. Moioli, P. A. Vargas, F.J. Von Zuben, and P. Husbands. Towards the evolution of an artificial homeostatic system. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, pages 4023–4030. IEEE Press, Piscataway, NJ, 2008.
35. F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli. The e-puck, a robot designed for education in engineering. In *In Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions (ROBOTICA)*, pages 59–65. Instituto Politecnico de Castelo Branco, Castelo Branco, Portugal, 2009.
36. H. Nakamura, A. Ishiguro, and Y. Uchilkawa. Evolutionary construction of behavior arbitration mechanisms based on dynamically-rearranging neural networks. In *Proceedings of Congress on Evolutionary Computation (CEC)*, pages 158–165. IEEE Press, Piscataway, NJ, 2000.
37. A. L. Nelson, G. J. Barlow, and L. Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4): 345–370, 2009.
38. S. Nolfi and D. Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press Cambridge, 2000.
39. S. Nolfi and D. Parisi. Evolving non-trivial behaviors on real robots: an autonomous robot that picks up objects. In *Proceedings of the 4th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 187–198. Springer, Berlin, Germany, 1995.
40. S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Proceedings of the 4th International Workshop on Artificial Life*, pages 190–197. MIT Press, Cambridge, MA, 1994.
41. C. W. Reynolds. Evolution of corridor following behavior in a noisy world. In *From Animals to Animats 3: Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior (SAB)*, pages 402–410. MIT Press, Cambridge, MA, 1994.
42. K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
43. E. C. Tolman and C. H. Honzik. Introduction and removal of reward, and maze performance in rats. *University of California Publications in Psychology*, 4:257–275, 1930.
44. A. B. L. Torta, M. A. Kramer, C. Thorn, D. J. Gibson, Y. Kubota, A. M. Graybiel, and N. J. Kopell. Dynamic cross-frequency couplings of local field potential oscillations in rat striatum and hippocampus during performance of a T-maze task. *Proceedings of the National Academy of Sciences*, 105(51): 20517–20522, 2008.
45. E. Tuci, V. Trianni, and M. Dorigo. 'Feeling' the flow of time through sensorimotor co-ordination. *Connection Science*, 16(4):301–324, 2004.
46. E. Tunstel. Mobile robot autonomy via hierarchical fuzzy behavior control. In *Proceedings of the 6th International Symposium on Robotics and Manufacturing (WAC)*, pages 837–842, New York, 1996. ASME Press.
47. R. A. Watson, S. G. Ficici, and J. B. Pollack. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Robotics and Autonomous Systems*, 39(1):1–18, 2002.
48. L. D. Whitley. Fundamental principles of deception in genetic search. In *Foundations of Genetic Algorithms*, pages 221–241. Morgan Kaufmann, 1991.