Learning to Cooperate: A Hierarchical Cooperative Dual Robot Arm Approach for Underactuated Pick-and-Placing

Sander De Witte, Thijs Van Hauwermeiren, Tom Lefebvre, and Guillaume Crevecoeur, *Member, IEEE*

***Abstract*—Cooperative multiagent manipulation systems allow to extend on the manipulative limitations of individual agents, increasing the complexity of the manipulation tasks the ensemble can handle. Controlling such a system requires meticulous planning of subsequent subtasks, queried to the individual agents, in order to execute the master task successfully. Real-time (re)planning is essential to ensure the task can still be achieved when subtasks execution suffers from uncertainty or when the master task changes intermittently requiring real-time reconfiguration of the plan. In this article, we develop a supervisory control architecture tailored to the cooperation of two robotic manipulators equipped with standard pick-and-place facilities in the plane. We control the planar position and orientation of an object using two underactuated manipulators so that only the position of the object can be controlled directly. The desired orientation follows from the accumulation of alternating relative angles. A time-invariant policy function is trained using deep reinforcement learning, which can determine a finite sequence of pick-and-place maneuvers to manipulate the object to a desired configuration. Two policy architectures are compared. The first uses the kinematic model to determine the final step, whilst the second policy makes this decision itself. The more information is given to the policy, the easier it trains. In return, it becomes less adaptable and loses some of its generalizability.**

***Index Terms*—Cooperation, multiagent manipulation, pick-and-place, reinforcement learning, robotics.**

I. Introduction

DESIGNING a single autonomous robot, adaptable to all circumstances, is not attainable or at least not an economical use of resources. Instead, complex tasks are better handled by a combination of multiple standard robots that cooperate [1]. The benefits of cooperation between agents are described by Tan [2] amongst others. For example, manipulation tasks with a higher complexity can be performed by using cooperative multiagent manipulation systems, thus overcoming the manipulative limitations of individual agents. This trend of cooperation can be seen in a variety of robotic applications, such as multi-UAV control systems [3], heterogeneous multirobot systems combining aerial and mobile robots [4], manipulation tasks [5], or assembly operations [6]. The cooperation of robots, among themselves and with humans, significantly increases the spectrum of automated tasks. The downside is clearly that the complexity of coordinating and controlling these systems also increases.

Meticulous planning of subsequent tasks, queried to the individual agents, is required in order to have a successful execution of the master task. This problem is closely related to task and motion planning (TAMP) [7]. TAMP problems encompass both discrete decisions, arising from the discrete task planning, as continuous motion plans and handle these in an integrated way. These methods can be applied to multirobot systems, where the decisions of which robot to query are optimized as well [8]. Standard TAMP approaches would perform a combined optimization of the discrete actions, the geometric switches, and the continuous motion plans. In this article, we consider control and planning of manipulation tasks in a hierarchical sense, with a high level planner that is agnostic to the low-level control. Information from each agent is known by the planner, resulting in a centralized control policy which enables collaborative agents. In contrast to TAMP, we decompose our discrete decisions, the discrete actions and geometric switches, and continuous motion plans completely. A supervisory control problem should then only query subtasks leaving execution to the individual robots. This control decomposition allows to significantly reduce the associated optimization problem benefiting real-time computation. Our approach enables us to perform all discrete decisions using a real-time policy. In that way, the system can be reactive to changes in the environment, a change of goal state, or simply poorly executed subtasks. Other methods, such as a random tree search through motion primitives, face considerable difficulty as a consequence of the associated combinatorial and, therefore, computational complexity.

Manuscript received 13 December 2021; revised 23 March 2022; accepted 29 April 2022. Date of publication 3 June 2022; date of current version 16 August 2022. Recommended by Technical Editor S. Jeon and Senior Editor X. Chen. This work was supported by the Flemish Government (AI Research Program) and the Flanders Make (Multi Systems Learning Control). *(Corresponding author: Sander De Witte.)*

The authors are with the Department of ElectroMechanical, Systems and Metal Engineering, Ghent University, B-9052 Zwijnaarde, Belgium, and also with the Core Lab EEDT Decision and Control, Flanders Make Strategic Research Centre for the Manufacturing Industry, 3920 Lommel, Belgium (e-mail: sander.dewitte@ugent.be; thijs.vanhauwermeiren@ugent.be; tom.lefebvre@ugent.be; guillaume.crevecoeur@ugent.be).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TMECH.2022.3175484.

Digital Object Identifier 10.1109/TMECH.2022.3175484

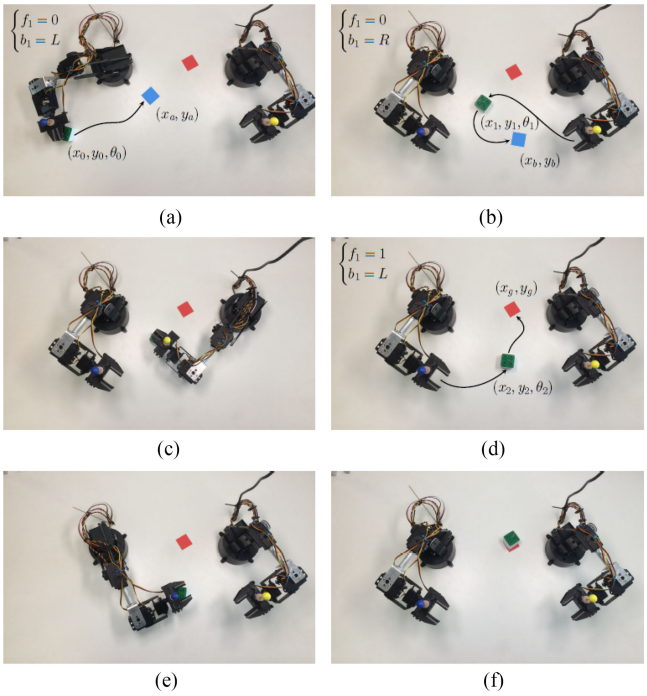


Fig. 1. Example of a possible action sequence resulting in the final configuration (x, y, θ) of the object (red) starting from a random initial configuration (green), with the intermediary configurations shown in blue. The orientation of the object cannot be controlled directly since the manipulators are only 5-DOF and therefore need to cooperate. The low-level control is concerned with the motion planning, performed by the individual robots, visualized with the black arrows. The supervisory control determines which action is taken next, revealing the hierarchical control architecture. (a) $a_1 = (0, L, x_a, y_a)$. (b) $a_2 = (0, R, x_b, y_b)$. (c) $a_2 = (0, R, x_b, y_b)$. (d) $a_3 = (1, L, x_g, y_g)$. (e) $a_3 = (1, L, x_g, y_g)$. (f) $(x_3, y_3, \theta_3) \approx (x_g, y_g, \theta_g)$.

We consider a particular manipulation problem formulation where two individual manipulators share a subset of their individual workspaces. We are motivated by the question of how this shared workspace can be used by the agents to collaborate in such a sense that they are able to execute complex tasks, exceeding their individual limitations. In this article, a setup consisting of two robotic manipulators tasked with maneuvering an object into an arbitrary goal configuration (position and orientation) in the global workspace is introduced. By working together, arbitrary goal configurations can be attained through multiple, collaborative actions in the shared workspace, illustrated in Fig. 1. To this end, a policy is determined that outputs the optimal sequence of action (including which robot to perform the manipulation) to achieve the goal. In order to cope with dynamic changes in the environment (change of goal configuration or poor executed subtasks) in real time, a time-invariant policy is pursued that depends only on the instantaneous object configuration and the desired goal configuration. Such a policy is obtained using deep reinforcement learning (DRL) [9]. DRL algorithms are applied in a number of robotic applications, to perform motion planning for robotic manipulators [10], to perform complex (dexterous) manipulation [11], [12], to train multiple agents [13], or to train end-to-end policies from an RGB camera image to control [14].

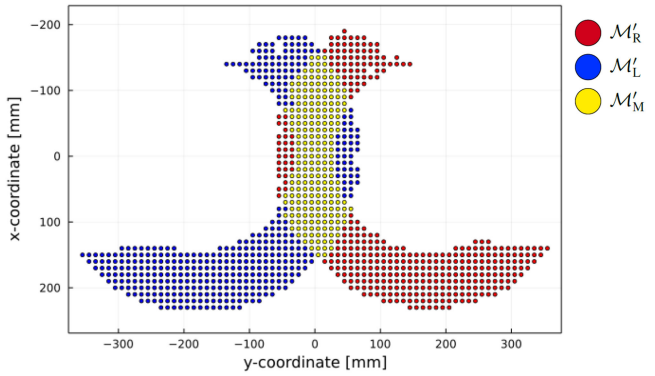


Fig. 2. Illustration of the discretized manipulation spaces ($\mathcal{M}'_{\mathrm{R}}$ and $\mathcal{M}'_{\mathrm{L}}$) with mutual manipulation space as the intersection in yellow ($\mathcal{M}'_{m}$) for the small-scale setup shown in Fig. 4.

In this article, we apply DRL on the system level to find a policy that determines the next action to be performed by a certain robot. The policy is trained in simulation, in order to reduce training time. Combining physical simulation with deep learning techniques shows to be advantageous since the large quantity of data needed for deep learning can be produced efficiently. Furthermore, it is safer and it is inexpensive [15], [16]. The validity of our solution is proven on an experimental setup (shown in Fig. 4).

The main contribution of this article is to overcome inherent underactuation in a multirobot manipulation problem by exploitation of a mutual workspace. We propose a solution that casts the problem as a sequential TAMP problem and decouple the TAMP level. This decomposition allows a supervisory control system to output the next best action (position in the mutual workspace). By decoupling the TAMP, the task planning (i.e., the optimal sequence) is done in real time. The policy used by this supervisory controller is trained using reinforcement learning in simulation. Learning cooperation this way results in a real-time adaptable policy by calculating the computational cost of the discrete decisions, resulting from cooperation, offline. Experimental validation proves that the method can be translated to a real setup.

II. Problem Statement

In this section, the considered small-scale setup is described together with the associated manipulation problem. As shown in Figs. 1 and 4, the setup consists of a work cell equipped with two plane symmetrical robotic manipulators and a vision system that can identify and locate objects in the workspace. Both robotic manipulators, henceforth referred to as the *left* (L) and *right* (R) manipulator, have five degrees of freedom (DoFs). Each individual manipulator is equipped with a lower level open-loop motion controller capable of executing pick-and-place maneuvers in the base XY-plane. We will refer to this plane as the manipulation plane. We use two DoFs to control the pitch and roll of the end-effector. The remaining three DoFs are used to control the end-effector's position in the manipulation plane, that is, its XY-coordinates with a fixed height. Consequently,

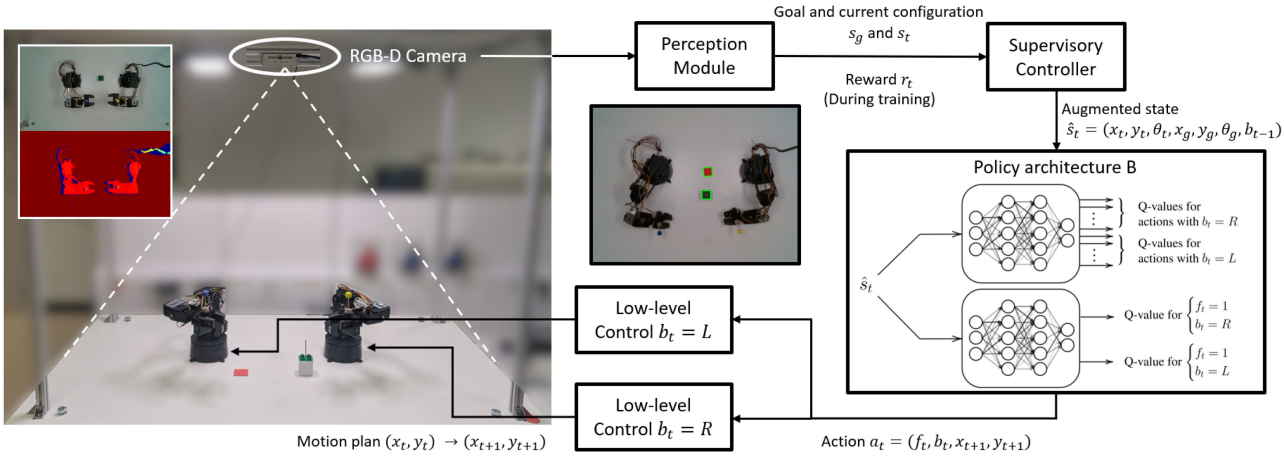


Fig. 3. Visualization of the deep Q-network with the final robot fixed (architecture A).

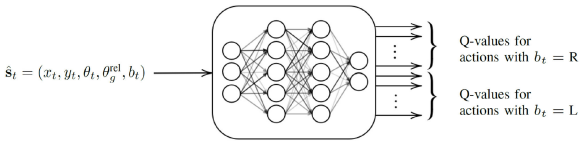


Fig. 4. Overview of the complete hierarchical structure in a block diagram. Architecture B is used for visualization, which can be seen in the represented DQN network.

it is not possible to control the end-effector's yaw, resulting in an underactuated manipulation problem. Because the yaw of the manipulated object cannot be controlled, its final orientation will depend on the respective pick-up and drop-off positions and the morphology of the manipulators. We refer to the individual manipulation spaces as $\mathcal{M}_{\text{L}} \subset \mathbb{R}^2$ or $\mathcal{M}_{\text{R}} \subset \mathbb{R}^2$, respectively, which are defined as the controllable set of end-effector configurations in the manipulation plane expressed as Cartesian coordinates with respect to a global frame of reference. Equivalently, the manipulation spaces are determined as the intersection of the individual manipulator workspaces and the manipulation plane. The feasible workspaces are attained as the collection of points that are collision-free, including self-collision and collision with the ground. The *global* $\mathcal{M}$ and *mutual* $\mathcal{M}_m$ manipulation spaces are defined, respectively, as the union and intersection of the individual workspaces (Fig. 2)

$$\mathcal{M} = \mathcal{M}_{\text{L}} \cup \mathcal{M}_{\text{R}}$$
$$\mathcal{M}_m = \mathcal{M}_{\text{L}} \cap \mathcal{M}_{\text{R}}.$$

We consider a pick-and-place task where objects are placed within the global manipulation space $\mathcal{M}$ and need to be repositioned to a new and given goal position, $(x_g, y_g) \in \mathcal{M}$. In our present context, a goal orientation θ_g of the objects is also specified. Adding the orientation to the manipulation space results in the observation space $\mathcal{S}$ since the orientation is only observable and uncontrollable. The manipulators ought to respect the desired orientation to complete the manipulation task successfully. The morphology and low-level steering of the individual manipulators is such that they cannot execute the pick-and-place task separately but must decide on an optimal cooperation strategy to achieve the desired object configuration

$$\mathbf{s}_g = (x_g, y_g, \theta_g) \in \mathcal{S} = \mathcal{M} \times \mathbb{R}.$$

The objective of this work is to find a supervisory control system that is capable of querying pick-and-place maneuvers to the individual manipulators for manipulating objects from an arbitrary starting position and orientation to a given desired configuration. A straightforward solution to overcome the limitations of the individual manipulation space $\mathcal{M}_{\text{L}}$ and $\mathcal{M}_{\text{R}}$ and gain access to the global manipulation space $\mathcal{M}$ is to agree upon a fixed intermediate point in the mutual manipulation space $\mathcal{M}_m$ and pass the object from manipulator to manipulator through that point. However, since the orientation cannot be controlled directly, a sequence of different hand-over positions must be determined such that the sum of subsequent relative reorientations accumulates into the desired orientation. An example is illustrated in Fig. 1.

III. Methodology

To solve this problem of realizing an underactuated pick-and-place under dynamic changes in the environment, we will resort to a learning strategy that learns to cooperate. Before doing so, we establish a mathematical problem formulation.

A. Problem Formulation

The state of the system is determined by $\mathbf{s}_t$ which contains the Cartesian coordinates of the *manipulated object* and its orientation. The goal configuration of the object is denoted as $\mathbf{s}_g$ and contains the same information as the state

$$\mathbf{s}_t = (x_t, y_t, \theta_t) \in \mathcal{S}$$
$$\mathbf{s}_g = (x_g, y_g, \theta_g) \in \mathcal{S}.$$

Formally, our supervisory control needs to determine the following decisions: 1) the agent to query (L or R), 2) whether the action is final, and 3) the next drop-off location. Since the

goal state's $\mathbf{s}_g$ location can be anywhere in the global manipulation space, including the mutual manipulation space $\mathcal{M}_m$, or if the goal configuration is changed mid-execution, it is not straightforward which manipulator to query. Therefore, these decisions are left to the supervisory control system. They are represented by the boolean $f_t \in \{0,1\}$ indicating whether the present action is final, and the discrete variable $b_t \in \{\mathrm{L},\mathrm{R}\}$, indicating whether to query the left or right manipulator. The next drop-off position, on the other hand, is represented by the actions $(x_{t+1}, y_{t+1}) \in \mathcal{M}$ in Cartesian coordinates. Note that the drop-off position is only relevant when $f_t = 0$; otherwise, the drop-off location is equivalent to the goal position by definition so that, in practice, $(x_{t+1}, y_{t+1}) \in \mathcal{M}_m$ when $f_t = 0$. Resulting in the following action space $\mathcal{A}$

$$\mathbf{a}_t = (f_t, b_t, x_{t+1}, y_{t+1}) \in \mathcal{A} = \{0,1\} \times \{\mathrm{L},\mathrm{R}\} \times \mathcal{M}.$$

The dynamics of the problem are governed by a nonlinear function that depends on the forward kinematics of the robotic manipulators. The kinematics depend on the previous state, the next position, the manipulator that is queried, and whether the query is final or not. The state update rule results from this nonlinear function and does not depend on the motion plan between two positions since the relative orientation between the object and the robot is assumed to be constant

$$\mathbf{s}_{t+1} = \mathbf{f}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_g) = (x_{t+1}(\mathbf{a}_t), y_{t+1}(\mathbf{a}_t), \theta(\mathbf{s}_t, \mathbf{a}_t)).$$

Our objective is, thus, to design a time-invariant policy function $\pi : \mathcal{S}^2 \mapsto \mathcal{A}$ so that for any initial state in the global manipulation space, a finite sequence pick-and-place maneuvers is determined that manipulate the object into the desired configuration. To design such a policy, we consider a first-exit optimal control problem formulation, which is an example of a Markov decision process (MDP)

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\mathbf{s}_0 \sim \mathcal{U}(\mathcal{S})} \left[\sum_{t=0}^{t'} r(\mathbf{s}_t, \mathbf{s}_g, \pi(\mathbf{s}_t, \mathbf{s}_g)) \right].$$

Here, $\mathcal{U}(\mathcal{S})$ denotes the uniform distribution on $\mathcal{S}$. The function $r : \mathcal{S}^2 \times \mathcal{A} \mapsto \mathbb{R}$ is defined as the reward and can be used to express abstract features of the policy such as maximum accuracy or minimal execution time. Next to the reward, a state and action are defined for the MDP. The action is as defined above and indicates the next position of the block, combined with what robot should perform the action, and the discrete decision if the action is final. The low-level motion plan, actually performed by the robots, is determined by each manipulator itself. The policy can terminate the sequence at any time t' by taking $f_t = 1$. Consequently, the length of each sequence will differ. Taking $f_t = 1$ initiates the final step, which can be determined by the policy itself or externally. The policy depends on the goal state, making it context dependent. Therefore, the same policy can be used even if the context changes, e.g., after a manipulation task is successfully completed or mid-execution.

B. Solution Strategy

Here, we detail the proposed solution strategy.

1) Discretization of $\mathcal{M}_m$: In order to downscale the solution space, the mutual workspace is discretized. Using the inverse kinematics of the robotic manipulators, we verify the feasibility of an arbitrary discrete set of points $\mathcal{D} \in \mathbb{R}^3$. This is done for both manipulators, resulting in two discretized manipulation spaces $\mathcal{M}'_R \subset \mathcal{M}_R$ and $\mathcal{M}'_L \subset \mathcal{M}_L$. Again the discretized mutual manipulation space can be obtained from the intersection of both discretized manipulation spaces, $\mathcal{M}'_m = \mathcal{M}'_{\mathrm{L}} \cap \mathcal{M}'_{\mathrm{R}}$. The resulting discretized mutual manipulation space $\mathcal{M}'_m \subset \mathcal{M}_m$ has a finite cardinality $M = \mathrm{card}(\mathcal{M}'_m)$. A 2-D visualization of this mutual workspace is shown in Fig. 2. As a result, $\mathcal{A}' = \{0,1\} \times \{\mathrm{L},\mathrm{R}\} \times \mathcal{M}'_m$.

2) Policy Definition: Since the problem is deterministic and the action space is now discrete, a possible strategy could be to tackle the problem using a shortest path algorithm such as Dijkstra or A*. As a result of the dimensionality of the state-space and the connectedness of the individual states, this would generate a densely connected graph. The supervisory policy ought to search this graph in real time since we desire it to be context dependent, making this method impractical. Therefore, instead, we opt to find a parameterized policy $\pi(\cdot,\cdot) \approx \pi(\cdot,\cdot;\phi)$ assigning a subtask each discrete time step to one of the manipulators, after which they perform their own low-level motion planning. Next, we propose two alternative architectures to determine such a policy. A distinction can be made based on the treatment of the final step. The final step indicates the termination of a sequence and, ideally, would result in the correct orientation at the goal location. This decision could be made by the policy or externally, e.g., based on the kinematics of the system, resulting in two policy architectures. In order to implement the dependency on the changing goal in the MDP, the state will have to be augmented to include information about the goal. This information changes based on how the final step is handled, resulting in two different augmented states for each architecture.

a) Architecture A (final robot fixed): The final action is defined by the boolean f_t. It affects the problem in a highly nonlinear fashion, rendering a hard to find optimal solution. This boolean can be eliminated by limiting the goal state to the manipulation space of one of two robotic manipulators, i.e., $(x_g, y_g) \in \mathcal{M}_{b_{\text{final}}} - \mathcal{M}_m$ where $b_{\text{final}} \in \{\mathrm{L},\mathrm{R}\}$ indicates the robot that has to perform the final action. When a goal state $\mathbf{s}_g$ is queried, it is then easily verified whether $b_{\text{final}} = \mathrm{L}$ or $b_{\text{final}} = \mathrm{R}$ which is then used to determine which policy to activate. We must, thus, use and train *two* policies depending on which manipulator needs to execute the final action. This design choice, thus, limits the applicability of the policy since the goal position is limited to a certain robot's manipulation space.[1] Now as soon as the final robot has been identified, we can check for each state $\mathbf{s}_t \in \mathcal{M}_m$ whether it corresponds to the goal configuration $\mathbf{s}_g$ when we would use the final manipulator to maneuver it to the goal position (x_g, y_g). This is implemented by representing the goal configuration as a desired relative orientation between the object and the unique final robot $\theta^{\text{rel}}_{\text{final}}$ which is possible since we use different policies depending on the robot that will be queried last. The relative orientation

[1] $\mathcal{M}_m$ could be included by arbitrarily assigning a manipulator as final robot.

between the object and the final robot can be calculated by taking the difference between the object's orientation θ_t and the yaw of the robot's end-effector, which is the result of the manipulator's morphology and the pick-up and drop-off position. Note that $\mathbf{g}$ depends on the manipulator

$$\theta_t^{\text{rel}} = \mathbf{g}_{b_{\text{final}}}(\mathbf{s}_t) = \theta_t - \theta_{b_{\text{final}}}(x_t, y_t).$$

During a pick-and-place, the relative orientation does not change since a rigid grasp is assumed. Hence, if θ_t^{rel} equals $\theta_g^{\text{rel}} = \mathbf{g}(\mathbf{s}_g)$, the object will have the correct orientation when placed at the goal position. The action space reduces to

$$\mathbf{a}_t = (b_t, x_{t+1}, y_{t+1}) \in \mathcal{A}'' = \{\text{L}, \text{R}\} \times \mathcal{M}'_m.$$

Now, the policy needs to find a sequence of actions resulting in a state in the mutual manipulation space $\mathcal{M}_m$ that indirectly corresponds to a certain goal state $\mathbf{s}_g \in \mathcal{M} - \mathcal{M}_m$. Whether the step is final is not dictated by the policy but is determined by an auxiliary routine that is based on θ_t^{rel}. Finally, the previous robot that performed an action is provided to the system as well. This gives the policy the chance to reason based on the previous robot, which makes it possible to impose a switch between the robots. This switch is necessary to change the relative orientation between the block and the robot. Resulting in the following augmented state

$$\hat{\mathbf{s}}_t = (x_t, y_t, \theta_t, \theta_g^{\text{rel}}, b_{t-1}) \in \mathcal{M} \times \mathbb{R}^2 \times \{\text{L}, \text{R}\}.$$

The reward received after acting on the system is sparse. A zero is received when the state, at time t and residing in the mutual workspace, results in the desired relative orientation with the given final robot. Any other step receives minus one

$$r = \begin{cases} 0, & \theta_t^{\text{rel}} = \theta_g^{\text{rel}} \\ -1, & \theta_t^{\text{rel}} \neq \theta_g^{\text{rel}} \end{cases}.$$

b) Architecture B (learned final step): A second approach is a policy that dictates when the final step is taken, resulting in a full action as defined in Section III-A. The final step is now part of the sequence, in contrast to the previous architecture. The policy will have to dictate from which state $s_t \in \mathcal{S}$ the object needs to be placed at the goal position and with what manipulator. Additionally, it will have to determine a sequence of actions to end up in that state. The goal state is provided in its entirety to the policy and the robot that placed the block at its position as well. The following augmented state is provided to the policy

$$\hat{\mathbf{s}}_t = (x_t, y_t, \theta_t, x_g, y_g, \theta_g, b_{t-1}) \in \mathcal{S}^2 \times \{\text{L}, \text{R}\}.$$

Introducing this final step in the action space introduces a number of possible terminations of an episode. Similar to the previous architecture, taking an action that does not result in the object ending up in the final configuration is punished, since the episode length needs to be minimized. If the policy tries to drop off or pick up an object at a position the manipulator b_t cannot reach, because this position is outside its manipulation space, a large cost is returned. Finally, a sequence can end in two ways: the object is placed at its goal position either with the goal orientation or with a different orientation. The former is rewarded with a large reward, whilst the latter receives a cost equal to the difference in orientation scaled by a constant. The experimental problem of the pick-and-place task by two robotic manipulators is solved by implementing a heuristic reward. After some reward shaping, the following reward function is received:

$$r = \begin{cases} 10, & (x_t, y_t) \equiv (x_g, y_g) \text{ and } |\theta - \theta_g| \leq \epsilon \\ -C \cdot |\theta_t - \theta_g|, & (x_t, y_t) \equiv (x_g, y_g) \text{ and } |\theta - \theta_g| > \epsilon \\ -1, & (x_t, y_t) \in \mathcal{M}_m \\ -25, & (x_t, y_t) \vee (x_{t-1}, y_{t-1}) \notin \mathcal{M}_{b_t} \end{cases}.$$

Here, $\mathcal{M}_{b_t}$ is the manipulation space of the robot that performs the action, indicated with the boolean $b_t \in \{\text{L}, \text{R}\}$. The final case occurs when the robot tries to pick up or drop off the object from or to a position outside its manipulation space. This can happen in the first step, if the object is outside the reach of the robot indicated by b_t, or in the final step, if the goal position is outside the robot's reach. The more complicated reward function indicates the difference in complexity between both architectures.

3) Reinforcement Learning: To determine the parameterized policy, DRL is used. The use of discrete actions steers us in the direction of Q-learning, where, for each action, the value function, estimating future rewards, is returned. Mnih *et al.* [17] propose the use of a neural network to estimate the value function. A neural network, called a deep Q-network (DQN), is trained to approximate the optimal action-value function $Q(\mathbf{s}, \mathbf{a}; \theta) \approx Q^*(\mathbf{s}, \mathbf{a})$ with weights $\boldsymbol{\phi}$ and

$$Q^*(\mathbf{s}, \mathbf{a}) = \max_{\pi} \mathbb{E}[R_t | \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}, \pi].$$

The future discounted return at time t is

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

where γ is the discount factor and determines the horizon.

This neural network is trained off-policy using a stochastic gradient descent. The required parametric policy is defined implicitly using the greedy strategy $\pi(\mathbf{s}) = \max_a Q(s, a; \boldsymbol{\phi})$. During training, the actions are selected using an ϵ-greedy strategy. A random action is selected with probability ϵ, whilst a greedy strategy is used with probability $1 - \epsilon$. The value of ϵ decreases exponentially during training. Q-learning updates are applied to batches of experience $(s, a, r, s') \sim U(\mathcal{D})$, where $U(\mathcal{D})$ denotes a uniform distribution over the set of experiences $\mathcal{D}$. The loss function used by the Q-learning update at iteration i is

$$L_i(\boldsymbol{\phi}_i) = \mathbb{E}_{(s,a,r,s') \sim U(\mathcal{D})}[(r + \gamma \max_{a'} Q(s,' a'; \boldsymbol{\phi}_i^-) - Q(s, a; \boldsymbol{\phi}_i))^2]$$

with $\boldsymbol{\phi}_i$ being the parameters of the Q-network at iteration i and $\boldsymbol{\phi}_i^-$ the network parameters used to compute the target at iteration i. The target is only updated every C steps.

The above can help to find the parameterized policies when considering 1) a fixed final robot or 2) a learned final step. The two different policy architectures result in a different neural network approximators.

In the former, the possible drop-off positions (x_{t+1}, y_{t+1}) can contain all points in the discretized mutual workspace $\mathcal{M}'_m$, i.e., all yellow positions in Fig. 2. The boolean b_t is implemented by doubling the mutual workspace. A function $f : \mathcal{M}'^2_m \mapsto \{0, 1\} \times \mathcal{M}'_m$ translates each position to a position performed by a certain robot, by looking at its index. The first half is performed by the first robot, whilst the second half by the second robot. The network is visualized in Fig. 3.

For architecture B, a final action needs to be introduced. Two additional actions are added, containing the final step done by one of the two robots. When this action is chosen, the object is placed at the goal position $(x_g, y_g) \in \mathcal{M}$ and the orientation θ is compared to the goal orientation θ_g. This action is final and terminates the episode. These two actions are visualized in Fig. 4 in the output of the DQN. The reason a parallel network is used to represent the Q-value for the two final actions is discussed in more detail in Section IV-B2. Here, two methods are introduced to find the policy, corresponding to architecture B.

4) Learning in Simulation: A large amount of episodes needs to be played to train the network. This is not practicable on a real setup, since this would take weeks to train. Luckily, the deterministic property of the setup and its open loop control make it possible to translate the problem to simulation. Even more, the policy can be trained without taking the motion planning into account since the state update rule does not depend on the motion plan, as mentioned previously. Only the robot poses at the drop-off and pick-up locations need to be calculated.

The trained policy in simulation will be executed on the real setup. This translation of simulated environment to real environment depends heavily on the models and the repeatability of the two robotic manipulators but is partially faced by the closed-loop behavior of the supervisory controller.

IV. Experiments

In this section, the training results in simulation are discussed first. Second, the translation to the real setup is considered.

A. Setup

Two low-cost robotic manipulators are used, two Lynxmotion AL5A robotic arms with five DOFs. The two manipulators can perform pick-and-place maneuvers on a square object by grasping the toothpick attached at the center. They are steered in an open-loop fashion from the pick-up position to the drop-off position. The path planning is currently implemented with a heuristic method that follows an approximately linear path in the feasible joint space, though keeping the square leveled, since this minimizes the displacement in joint coordinates. Collision avoidance is considered both with the ground, as with itself; collision with the other robot is avoided by limiting one robot at a time in the mutual work space. The path planning could be implemented with any existing trajectory optimizer or motion planner. A camera (Intel RealSense Depth Camera D435) is used to perceive information from the system, i.e., configuration of the block and the goal.

A picture of the setup is shown in Fig. 4. Two lights, which can be seen at the top of the frame next to the camera, are mounted in order to provide consistent lighting conditions for the vision. The machine vision is based on color and edge detection, making it possible to extract the four corners of the object and the goal. From these corners, the orientation with regard to the global frame of reference can be calculated.

TABLE I
Used Hyperparameters for the Different Trained Policies

Name		optimiser	Learning rate	DNN size	Steps
Architecture A		Adam	0.001	5|64|354	150,000
Architecture B	No pretraining	Adam	0.001	7|512|512|356	500,000
	Pretraining: partial network	Adam	0.001	7|256|256|2	250,000
	Pretraining: full network	Adam	0.001	7|256|256|354 7|256|256|2	150,000
	Final training with pretraining	Adam	0.001	7|256|256|354 7|256|256|2	1,750,000

The DNN size is indicated by $a|b|c$, where a indicates the number of inputs, b indicates the number of hidden notes, and c indicates the number of outputs. A parallel network is represented by two networks stacked.

B. Implementation

All code is written in Julia [18]. The DQN network is trained using the Julia package "ReinforcementLearning.jl."

1) Discretization: A 3-D grid of points spaced 10 mm apart in each direction is taken as the set of discrete points $\mathcal{D}$. The feasibility of all points in this set is checked for each robotic manipulator resulting in two point clouds, corresponding to the discrete manipulation spaces. The discretized mutual manipulation space is taken as the intersection of these two point clouds. A 2-D section of the calculated discretized manipulation spaces are shown in Fig. 2.

2) Networks and Training: A selection of the used hyperparameters are shown in Table I. The sigmoid function is used as activation function and Adam [19] as optimizer. Three methods are compared. The first method corresponds to the policy architecture A, with a fixed final robot. The last two methods use policy architecture B and consist of a method without pretraining and one with pretraining. The latter uses the network proposed in Fig. 4. This is done so that the network controlling the two final actions can be pretrained. Pretraining is done to improve the learning rate and find a viable solution, which will be shown in the next section. The network is first trained on a one-step environment. The goal state corresponds to the start state, such that if the object is placed directly at the goal position with the correct robot, the object will have the goal orientation. This pretraining is done in two steps, first with the parallel network only, with two outputs, and finally with the complete network.

C. Results

All architectures are trained in simulation and subsequently transferred to the real setup. No further training on the setup has been performed.

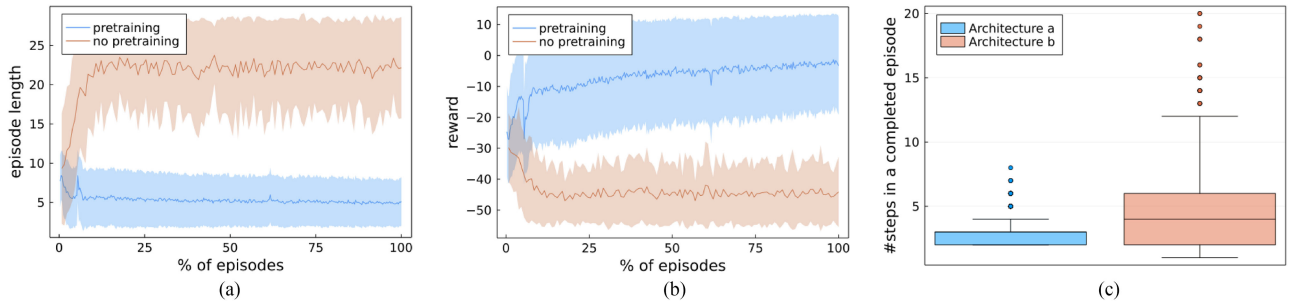


Fig. 5. (a) and (b) Reward and episode length during training compared for the methods with and without pretraining. (c) Visualization of results (#steps) for 1000 unseen experiments with the trained policies, comparing the performance for both architectures in simulation. (a) Episode lengths during training. (b) Rewards. (c) Episode lengths after training.

TABLE II
PERFORMANCE PARAMETERS FOR BOTH NETWORKS TESTED FOR 1000 RANDOM EPISODES IN SIMULATION

	Success ratio [%]	Average number of steps
Architecture A	98.8	2.78
Architecture B	71.4	4.54

TABLE III
PERFORMANCE PARAMETERS FOR BOTH NETWORKS TESTED FOR 100 RANDOM EPISODES ON THE REAL SETUP

	Success ratio [%]	Average number of steps
Architecture A	90	3.67
Architecture B	72	4.83

1) Importance of Pretraining: First, we elaborate on the effect of the previously discussed pretraining. The network has to decide when to perform the final step; it has to learn what state in the mutual work space corresponds to a certain final configuration. This dependency is highly nonlinear and therefore difficult to train. On top of this, the two final actions, or two final outputs in the DQN, encompass only a small part of the 356 outputs. When using an ϵ-greedy strategy, where actions at the start have a higher chance to be chosen at random, these two actions will have a low probability of being chosen. By pretraining, we force the network to learn this dependency first, which results in a higher reward for these two actions.

The maximum episode length during training is limited to 25 steps. The episode can end in less steps, if the final action is chosen earlier. The effect of this pretraining is clearly visible in Fig. 5(a) and (b) since the method using no pretraining does not find a viable solution. The figures show the mean and variance per 500 episodes. The network trained with pretraining converges to an average episode length of five steps, whilst the network trained without pretraining settles around the maximum episode length. The reward, on the other hand, is still rising for the method with pretraining, whilst it converges to a low value for the method using no pretraining.

In the subsequent section, only the method with pretraining is used to compare the policy architectures and is, thus, simply referred to as architecture B from now on.

2) Comparison of the Two Architectures: The two architectures are compared on the basis of two performance indicators: the success ratio and the number of steps in an episode. The former is defined as the percentage of episodes for which the goal configuration is reached. The policies are tested on 1000 random start and goal states; the results are shown in Table II.

A box plot showing the number of steps in more detail is shown in Fig. 5(c). The first architecture outperforms the second, with fewer steps and a higher success ratio. However, it is less applicable since the goal state is limited to the manipulation space of the final robot, which is fixed.

The final architecture can be looked at in more depth, which is done in Fig. 6(a). The episode can end four ways: at the goal position with the desired orientation (cr), at the goal position with the wrong orientation (crwa), goal position not reachable by the final robot (wr), or the episode could end without trying to place it at the goal position (nf). It can be seen that in the majority of the cases, the robot places the object at the goal position with the correct robot. When placed at the goal position, the correct orientation is achieved most of the time, indicating the policy has found a viable action sequence before calling the (correct) final action.

3) Experimental Validation: Both policies trained in simulation are tested on a real setup. No further training is done on the setup. An example of a sequence performed on the setup is shown in Fig. 1. The position and the orientation of the block are updated in real time, after each discrete time step. The performance parameters for both robots are found in Table III. The error on the orientation is roughly the same for both policy architectures, with an average of 4.56° and 5.15° and a standard deviation of 4.78° and 5.12°, respectively, for architectures A and B.

Similar to the results in simulation, the first policy needs fewer steps than the second to end up with the correct orientation, although the difference becomes smaller. Comparing Figs. 5(c) and 6(c) shows that the first is less performant on the real setup, whilst the second shows similar results in simulation as on the real setup. Presumably, the latter translates better to the real world because it does not use the robotic manipulator's model to take a decision on the final step. Therefore, it is less sensitive to errors in placing the block.

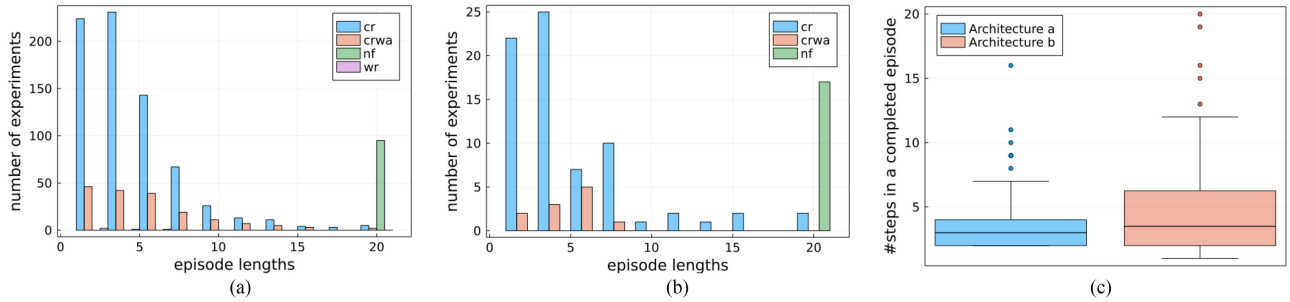


Fig. 6. (a) and (b) In-depth look for architecture B, both in simulation as on the real setup. (c) Comparison of both architectures on the real setup. (a) Simulation. (b) Real setup. (c) Episode lengths.

Fig. 6(b) shows the distribution of the episode lengths for architecture B. Since, in this, no wrong robot is called upon to perform the final step, only three possible cases are indicated. The results are similar in both figures, although the second figure shows less of a normal distribution which could be attributed to the lower number of experiments.

Translating the policy to the real setup seems to work since similar results are achieved. This could be attributed to the fact that we work on a high level, making the exact model of the system less important. Using the hierarchical control structure is beneficial to react to uncertainties in the environment, as a result of the stochasticity of the real world. Additionally, since a certain error ϵ is allowed both in simulation and in the real world, the model does not need to be exact. Using the policy in a closed loop aids the translation as well since the previous errors in the model are of no importance. The policy finds a new action based on the new state, which is updated each step.

V. Conclusion

A first step is made to have a multirobot system that is capable of detecting opportunities to collaborate in an autonomous fashion. Although a successful cooperative multiagent manipulation system is achieved, it is not yet fully autonomous. The trained hierarchical control system works in real time and is able to adapt to a dynamic environment. The goal state could be changed mid-execution or the block could be moved in between steps. This is achieved by training a time-invariant parameterized policy in simulation, using reinforcement learning techniques. This principle can be extended to include more robots or to have a larger variety of possible actions. No information of the goal configuration is used in the second policy architecture; it is only checked if the goal configuration is achieved. Therefore, a different type of goal configuration could be considered in future work.

Training in simulation proves to be beneficial by allowing more iterations in less time. With some additional hyperparameter tuning, the performance and learning rate of the second architecture could be improved. Others algorithms could be looked at as well, e.g., the sample efficiency could be improved by using hindsight experience replay (HER) [20]. The benefit of training in simulation becomes clear when comparing numbers. Validating 100 episodes on the real setup, with a maximum of 20 steps per episode, took 3.5 h with an average of around 7.5 steps. Performing the 1 750 000 steps needed to train the second method would take 8167 h or a little more than 340 days nonstop; this is without taking into account the pretraining.

Translating the policy trained in simulation to the real-world works but has its limitations. When translated to the real setup, providing more information to the system works less well, since the discrepancy between the simulation and the real robotic manipulators are embedded in the system. This possible overfit on deterministic data in simulation can be further explained by the use of partially discrete action and state space. Although the policy expects a certain exact position, the actual position might differ. This actual position is not seen before by the policy since a discrete set of actions is taken in simulation. Translating the policy to the real setup could be improved by describing the action and state space by continuous variables. Additionally, by using the continuous representation, the system could become more adaptable by, e.g., being able to overcome certain constraints in the workspace. The continuous description of the mutual work space could be adapted to include time varying constraints as well, making the method applicable in an environment containing moving obstacles, or possibly a human-centric environment. A final extension, which would be possible with this representation, could be to include mobile robots.

Acknowledgment

The authors would like to thank Kamiel Vandewoude for his initial work in constructing the setup for experimental validation and providing us with helpful insights at the start of this research.

References

[1] Y. Rizk, M. Awad, and E. W. Tunstel, "Cooperative heterogeneous multi-robot systems: A survey," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 1–31, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3303848

[2] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proc. 10th Int. Conf. Mach. Learn.*, 1993, pp. 330–337.

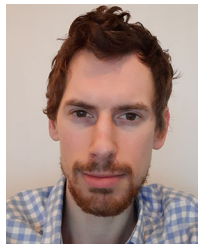
[3] H. Duan and D. Zhang, "A binary tree based coordination scheme for target enclosing with micro aerial vehicles," *IEEE/ASME Trans. Mechatronics*, vol. 26, no. 1, pp. 458–468, Feb. 2021.

[4] N. Bezzo *et al.*, "A cooperative heterogeneous mobile wireless mechatronic system," *IEEE/ASME Trans. Mechatronics*, vol. 19, no. 1, pp. 20–31, Feb. 2014.

[5] W. Gueaieb, F. Karray, and S. Al-Sharhan, "A robust hybrid intelligent position/force control scheme for cooperative manipulators," *IEEE/ASME Trans. Mechatronics*, vol. 12, no. 2, pp. 109–125, Apr. 2007.
[6] M. Dogar, A. Spielberg, S. Baker, and D. Rus, "Multi-robot grasp planning for sequential assembly operations," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2015, pp. 193–200.
[7] C. R. Garrett *et al.*, "Integrated task and motion planning," *Annu. Rev. Control, Robot., Auton. Syst.*, vol. 4, no. 1, pp. 265–293, 2021. [Online]. Available: https://doi.org/10.1146/annurev-control-091420-084139
[8] I. Umay, B. Fidan, and W. Melek, "An integrated task and motion planning technique for multi-robot-systems," in *Proc. IEEE Int. Symp. ROSE*, 2019, pp. 1–7.
[9] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017.
[10] M. Kim, D.-K. Han, J.-H. Park, and J.-S. Kim, "Motion planning of robot manipulators for a smoother path using a twin delayed deep deterministic policy gradient with hindsight experience replay," *Appl. Sci.*, vol. 10, 2020, Art. no. 575.
[11] H. Nguyen and H. La, "Review of deep reinforcement learning for robot manipulation," in *Proc. 3rd IEEE Int. Conf. Robotic Comput.*, 2019, pp. 590–595.
[12] A. Rajeswaran *et al.*, "Learning complex dexterous manipulation with deep reinforcement learning and demonstrations," in *Proc. Robotics: Sci. Syst.*, 2018, Art. no. 49, doi: 10.15607/RSS.2018.XIV.049.
[13] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, "Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications," *IEEE Trans. Cybern.*, vol. 50, no. 9, pp. 3826–3839, Sep. 2020.
[14] A. Singh, L. Yang, C. Finn, and S. Levine, "End-to-end robotic reinforcement learning without reward engineering," in *Proc. Robotics: Sci. Syst.*, 2019, Art. no. 73, doi: 10.15607/RSS.2019.XV.073.
[15] S. Höfer *et al.*, "Sim2real in robotics and automation: Applications and challenges," *IEEE Trans. Automat. Sci. Eng.*, vol. 18, no. 2, pp. 398–400, Apr. 2021.
[16] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," in *Proc. IEEE Symp. Ser. Comput. Intell.*, 2020, pp. 737–744.
[17] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://dx.doi.org/10.1038/nature14236
[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671
[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980
[20] M. Andrychowicz *et al.*, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30. Red Hook, NY, USA: Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf

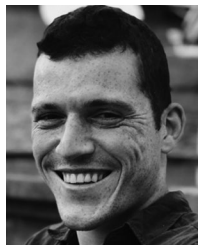
Sander De Witte received the master's degree in electromechanical engineering in 2021 from Ghent University, Ghent, Belgium, where he is currently working toward the Ph.D. degree in electromechanical engineering with the Department of Electromechanical, Systems and Metal Engineering.

His main research interests include robotic control and optimization in combination with learning techniques.



Thijs Van Hauwermeiren received the B.Eng. and M.Eng. degrees in industrial engineering from KU Leuven, Leuven, Belgium, in 2014 and 2015, respectively, and the M.Eng. degree in engineering from Ghent University, Ghent, Belgium, in 2018.

He is currently a Researcher with the Department of Electromechanical, Systems and Metal Engineering, Ghent University with a strong interest in model-based control and learning control in the context of (underactuated) robotics systems.



Tom Lefebvre received the M.Sc. degree in control engineering and automation and the Ph.D. degree in electromechanical engineering from Ghent University, Ghent, Belgium, in 2015 and 2019, respectively.

Since 2019, he has been a Postdoctoral Research Assistant. He is currently an Affiliate Member of Flanders Make, Flanders, Belgium, the strategic research centre for the manufacturing industry. His main research interests include foundational work on numerical methods for stochastic optimal control, gradient and stochastic trajectory optimization, and uncertainty quantification.

Guillaume Crevecoeur (Member, IEEE) received the master's and Ph.D. degrees in engineering physics from Ghent University, Ghent, Belgium, in 2004 and 2009, respectively.

He received a Research Foundation Flanders postdoctoral fellowship in 2009 and was appointed as Associate Professor with Ghent University in 2014. He is a member of Flanders Make, Flanders, Belgium, in which he leads the Ghent University activities on sensing, monitoring, control, and decision-making. With his team, he conducts research at the intersection of system identification, control, and machine learning for mechatronic and industrial robotic systems. His goal is to endow physical dynamic systems with improved functionalities and capabilities when interacting with uncertain environments, other systems, and humans.