Model Checking for Adversarial Multi-Agent Reinforcement Learning with Reactive Defense Methods

Dennis Gross[1], **Christoph Schmidl**[1], **Nils Jansen**[1], **Guillermo A. Pérez**[2]

[1] Institute for Computing and Information Sciences, Radboud University
[2] Department of Computer Science, University of Antwerp

Abstract

Cooperative multi-agent reinforcement learning (CMARL) enables agents to achieve a common objective. However, the safety (a.k.a. robustness) of the CMARL agents operating in critical environments is not guaranteed. In particular, agents are susceptible to adversarial noise in their observations that can mislead their decision-making. So-called denoisers aim to remove adversarial noise from observations, yet, they are often error-prone. A key challenge for any rigorous safety verification technique in CMARL settings is the large number of states and transitions, which generally prohibits the construction of a (monolithic) model of the whole system. In this paper, we present a verification method for CMARL agents in settings with or without adversarial attacks or denoisers. Our method relies on a tight integration of CMARL and a verification technique referred to as model checking. We showcase the applicability of our method on various benchmarks from different domains. Our experiments show that our method is indeed suited to verify CMARL agents and that it scales better than a naive approach to model checking.

Introduction

Deep cooperative multi-agent reinforcement learning (CMARL) is a powerful tool to handle sequential decision-making problems. It has improved the performance of various applications in critical domains like manufacturing, transportation, and resource allocation (Serrano-Ruiz, Mula, and Poler 2021; Qin et al. 2021; Zong and Luo 2022). CMARL consists of multiple agents, where each one learns a near-optimal policy based on a given common objective by making observations and gaining rewards through interactions with the environment (Wong et al. 2021). Compared to standard RL, CMARL may show superior performance and scalability in certain settings (Zhang et al. 2022).

MARL Problem

Despite the success of CMARL, the potential unsafe behavior of agents (Shalev-Shwartz, Shammah, and Shashua 2016; Amodei et al. 2016) and the security risk of adversarial attacks against critical infrastructures (Dablain 2017) limit its usage. Adversarial attacks introduce noise into the observations and may mislead the decision-making of the agents (Lin et al. 2020; Huang et al. 2017; Ilahi et al. 2022; Moos et al. 2022; Fujimoto and Pedersen 2021). Denoisers defend CMARL agents against adversarial attacks by cleaning the observations from the noise before they are perceived by the agents (Ohashi et al. 2021; Vincent et al. 2008; Bengio et al. 2013; Im et al. 2017; Bakhti et al. 2019; Serban and Poll 2018). However, these denoisers can make reconstruction mistakes and may recover wrong states. Until now, the performance of CMARL agents equipped with denoisers was measured by the decrease in the cumulative rewards of the CMARL system. Unfortunately, rewards lack the expressiveness to encode complex safety requirements (Vamplew et al. 2022; Hasanbeig, Kroening, and Abate 2020). For instance, with rewards, it is possible to determine the probability that all trains will arrive at their destination. However, rewards are not sufficient for other properties, for instance, the probability that trains will arrive in a specific order. *Model checking* (Baier and Katoen 2008) is not limited by properties that can be expressed by rewards (Hahn et al. 2019; Hasanbeig, Kroening, and Abate 2020; Vamplew et al. 2022), but supports a broader range of properties that can be expressed by *probabilistic computation tree logic* (PCTL; Hansson and Jonsson 1994).

Model checking is a formal verification technique that uses mathematical models to verify the correctness of a system with respect to a given property. Naive monolithic model checking is called ”naive” because it does not take into account the complexity of the system or the number of possible states it can be in, and it is called ”monolithic” because it treats the entire system as a single entity, without considering the individual components of the system or the interactions between them.

Contribution

This paper aims to allow the model checking of CMARL agents (equipped with denoisers) in an (adversarial) CMARL setting, which guarantees that CMARL agents still comply with given safety requirements. To achieve this, we take advantage of the fact that a CMARL system can be modeled as a *Markov decision process (MDP)* by treating the collection of agent actions as one *joint action* and representing all the RL agents via a *joint agent* (Boutilier 1996). We present a model checking method that allows us to verify CMARL agents. We evaluated our method on different benchmarks from the CMARL community (Competitions 2021) and model checking community (Hartmanns et al. 2019), and compared our approach with naive monolithic model checking.

To summarize, our **main contributions** are model checking methods and extensive benchmarking for (1) CMARL agents, (2) under adversarial attacks, (3) equipped with denoisers, and (4) the combination of (1), (2), and (3).

The paper is structured in the following way. First, we summarize the related work and position our paper. Second, we explain the fundamentals of our technique, third, we present the adversarial attack and defense setting and describe how to model check CMARL systems. Finally, we evaluate our method in multiple environments.

Related Work

There already exist multiple model checking approaches and case studies for multi-agent environments (Kwiatkowska et al. 2021, 2020; Junges et al. 2018; Chen et al. 2011; Bertrand and Fournier 2013; Kwiatkowska, Norman, and Parker 2019). The major difference to our work is that they do not focus on CMARL. Schuppe and Tumova propose a decentralized solution to a high-level task-planning problem for a multi-agent system under a set of possibly dependent LTL specifications (Schuppe and Tumova 2021). Lomuscio and Pirovano developed a parameterized verification method for checking unbounded probabilistic multi-agent systems against strategic properties. They do this by creating an abstract model whose states have two components: the first captures the state of the first m agents, and the second records the set of states that arbitrarily many other agents are in (Lomuscio and Pirovano 2020). In our case, the number of agents in the system is fixed at design time, we focus on CMARLs, and our agents do not necessarily share the same observations and actions. Furthermore, we distinguish between swarm systems and CMARL systems since, in swarm systems, many identical agents interact with each other to achieve a common goal (Hüttenrauch, Sosic, and Neumann 2019). CMARL agents can be different. Mqirmi et al. present a methodology that combines formal verification with (deep) RL algorithms to guarantee the satisfaction of formally-specified safety constraints in training and testing. They first use bisimulation to create an abstraction of the multi-agent system. Then they build a shield that restricts the agents' actions (Mqirmi, Belardinelli, and León 2021). We directly induce the agents into the formal model and verify its PCTL properties. Riley et al. introduce a new approach (they build upon their previous research (Riley et al. 2021a)) that combines CMARL with a formal verification technique termed quantitative verification. Their approach consists of four stages. First, they acquire information about the CMARL environment. Second, they model the environment as a formal PRISM model. Third, they synthesize policies for the formal PRISM model that guarantees given properties. Fourth, they learn CMARL agents in the non-abstracted environment, where the synthesized policies constrain learning (Riley et al. 2021b). In comparison, we train our CMARL agents and verify them directly in the modeled environment. Khan et al. present a CMARL approach to goal assignment and guaranteed collision-free trajectory planning for unlabeled robots operating in obstacle-filled 2D spaces. To ensure an agent still has a collision-free trajectory, they use an analytical model based on a policy that runs in the background and checks if the target velocities produced by the agents are safe (Khan et al. 2019). Wang et al. present a formal framework for collision avoidance in multi-robot systems, wherein an existing policy is modified in a minimally invasive fashion to ensure safety (Wang, Ames, and Egerstedt 2016). On the other hand, we allow the model checking of a broad range of CMARL systems.

Background

We now introduce the fundamentals of our work.

Probabilistic Systems

A *probability distribution* over a set X is a function $\mu : X \to [0,1]$ with $\sum_{x \in X} \mu(x) = 1$. The set of all distributions on X is denoted $Distr(X)$.

Definition 1 (Markov Decision Process)**.** *A Markov decision process (MDP) is a tuple $M = (S, s_0, A, T, rew)$ where S is a finite, nonempty set of states, $s_0 \in S$ is an initial state, A is a finite set of actions, $T: S \times A \to Distr(S)$ is a probability transition function. We employ a factored state representation where each state s is a vector of features $(f_1, f_2, ..., f_n)$ where each feature $f_j \in \mathbb{Z}$ for $1 \leq i \leq n$ (n is the dimension of the state). $rew\colon S \times A \to \mathbb{R}$ is a transition-reward function.*

The available actions in $s \in S$ are $A(s) = \{a \in A \mid T(s,a) \neq \bot\}$. An MDP with only one action per state ($\forall s \in S : |A(s)| = 1$) is a discrete-time Markov chain (DTMC). A path of an MDP M is an (in)finite sequence $\tau = s_0 \xrightarrow{a_0, r_0} s_1 \xrightarrow{a_1, r_1} ...$, where $s_i \in S$, $a_i \in A(s_i)$, $r_i := rew(s_i, a_i)$, and $T(s_i, a_i)(s_{i+1}) \neq 0$. A state s' is reachable from state s, if there exists a path τ from state s to state s'.

Definition 2 (Policy)**.** *A memoryless deterministic policy for an MDP $M = (S, s_0, A, P, rew)$ is a function $\pi\colon S \to A$ that maps a state $s \in S$ to action $a \in A$.*

Applying a policy π to an MDP M yields an induced DTMC, where all non-determinism is resolved. We specify the properties of a DTMC via the specification language PCTL (Wang et al. 2020).

Definition 3 (PCTL Syntax)**.** *Let AP be a set of atomic propositions. The following grammar defines a state formula: $\Phi ::= true \mid a \mid \Phi_1 \land \Phi_2 \mid \neg\Phi | P_{\bowtie p} | P^{max}_{\bowtie p}(\phi) \mid P^{min}_{\bowtie p}(\phi)$ where $a \in AP$, $\bowtie \in \{<, >, \leq, \geq\}$, $p \in [0,1]$ is a threshold, and ϕ is a path formula which is formed according to the following grammar $\phi ::= X\Phi \mid \phi_1 \; U \; \phi_2 \mid \phi_1 \; F_{\theta_i t} \; \phi_2 \mid G\,\Phi$ with $\theta_i = \{<, \leq\}$.*

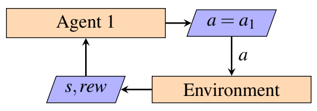
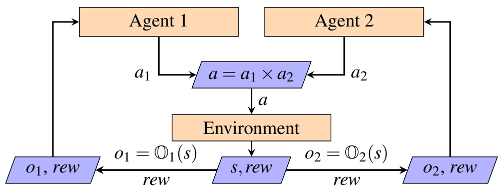
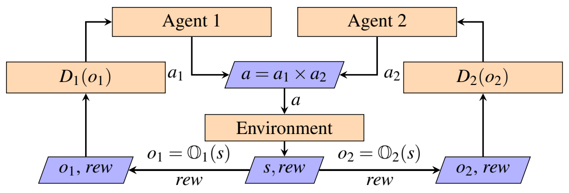


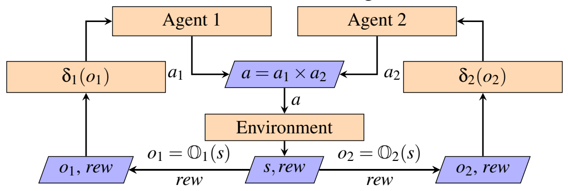
Figure 1: An MDP with one agent.



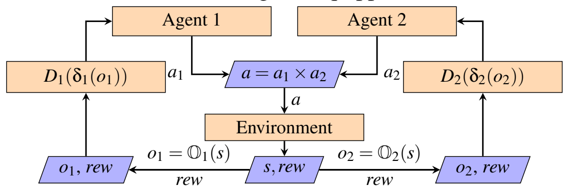
(a) CMMDP with two agents.



(b) CMMDP with two agents equipped with denoisers.



(c) CMMDP with two agents under attack.



(d) CMMDP with two agents equipped with denoisers under attack.

Figure 2: Different CMARL settings with and without denoisers $D_i(o_i)$ and with and without attacks $\delta_i(o_i)$. It assumes a setting with two agents and a joint action $a = a_1 \times a_2$. This action a is deployed to the environment and executed. The resulting state s and *rew* is separated into the observations.

For MDPs, PCTL formulae are interpreted over the states of the induced DTMC of an MDP and a policy. In a slight abuse of notation, we use PCTL state formulas to denote probability values. That is, we sometimes write $P_{\bowtie p}(\phi)$ where we omit the threshold p. For instance, in this paper, $P(F\ collision)$ denotes the reachability probability of eventually running into a collision. There exist a variety of model checking algorithms for verifying PCTL properties (Courcoubetis and Yannakakis 1988, 1995). PRISM (Kwiatkowska, Norman, and Parker 2011) and Storm (Hensel et al. 2022) offer efficient and mature tool support for verifying probabilistic systems.

Definition 4 (CMMDP)**.** *A cooperative multi-agent Markov decision process (CMMDP) is a tuple* $(S, s_0, I, A := \{A_i\}_{i\in I}, T, rew, \{O_i\}_{i\in I}, \{\mathbb{O}_i\}_{i\in I})$ *where S is a finite, nonempty set of states; $s_0 \in S$ is an initial state; I is a finite, nonempty set of agents; A_i is a finite, nonempty set of actions available to agent i; $T: S \times A_1 \times \ldots \times A_i \rightarrow [0,1]$ is a transition function, and $rew: S \rightarrow \mathbb{R}$ is a joint reward function. Each agent i gets its local partial observation $o_i \in O_i$ according to the observation function $\mathbb{O}_i(s): S \rightarrow O_i$. An observation $o_i \in O_i$ is a vector composed of features f_j from state s. The observations of all agents result in the full state.*

Each agent $i \in I$ has a policy $\pi_i: O_i \rightarrow A_i$ that maps an observation $o_i \in O_i$ to an action $a_i \in A_i$. The *joint policy* π induced by the set of agent policies $\{\pi_i\}_{i\in I}$ is the mapping from states into actions and transforms the CMMDP into an MDP (compare Figure 1 with Figure 2a). This is only possible if, for every state s and action a, the sub-policies π_i get a set of observations O_i that "reveals" the next state. They don't have to know the full state, but their combination should have that property. Inducing a joint policy π into an MDP yields an induced DTMC (Boutilier 1996).

Adversarial Multi-Agent Reinforcement Learning

We now introduce CMARL, *adversarial attacks* and *denoisers*. The standard learning goal for RL is to find a policy π in an MDP such that π maximizes the expected discounted reward, that is, $\mathbb{E}[\sum_{t=0}^{L} \gamma^t R_t]$, where γ with $0 \leq \gamma \leq 1$ is the discount factor, R_t is the reward at time t, and L is the total number of steps. CMARL extends the RL idea to find the near-optimal policies π_i in a CMMDP setting. Each agent policy π_i is represented by a neural network. A neural network is a function parameterized by weights θ_i. The neural network policy π_i can be trained by minimizing a sequence of loss functions $J(\theta_i, o_i, a_i)$ (Mnih et al. 2013).

Definition 5 (Adversarial Attacks)**.** *An adversarial attack $\delta_i: O_i \rightarrow O_i$ maps an observation o_i to another $o_{i,adv}$. An attack is α-bounded if $\|\delta_i(o_i) - o_i\|_\infty \leq \alpha$ with l_∞-norm defined as $\|\delta_i(o_i) - o_i\|_\infty = max_{\delta_{ik}\in\delta_i}|\delta_{ik} - o_{ik}|$. See Figure 2c for a visual example.*

In this work, we focus on the commonly used FGSM attack (Huang et al. 2017). Given the weights θ_i of the neural network policy π_i and a loss $J(\theta_i, o_i, a_i)$ with observation o_i and $a_i := \pi_i(o_i)$, the FGSM, denoted as $\delta_i: O_i \rightarrow O_i$, adds noise whose direction is the same as the gradient of the loss $J(\theta_i, o_i, a_i)$ w.r.t the state o_i to the state o_i and the noise is scaled by $\alpha \in \mathbb{Q}$ (see Equation (1)). We specify the ∇-operator as a vector differential operator. Depending on the gradient, we either add or subtract α.

$$\delta_i(o_i) = o_i + \alpha \cdot sign(\nabla_{o_i} J(\theta_i, o_i, a_i)) \tag{1}$$

To attack a CMARL system, we attack each RL policy separately via Equation (1).

Definition 6. *We denote a denoiser by $D_i: O_i \rightarrow O_i$. A denoiser D_i denoises an adversarial attack δ_i by passing the adversarial observation into the denoiser $D_i(\delta(o_i))$. A denoiser is successful if $\pi_i(D_i(\delta_i(o_i))) = \pi_i(o_i)$. An adversarial attack is successful if $\pi_i(D_i(\delta_i(o_i))) \neq \pi(o_i)$. See Figure 2b and Figure 2d for a visual example.*

A denoiser uses a neural network that gets trained by minimizing the loss function $J(\theta_i, \delta_i, o_i)$ (Bakhti et al. 2019; Serban and Poll 2018).

Methodolodgy

Safety and security need to be analyzed together. The reason for that lies in the fact that defense methods (like denoisers) can make mistakes that influence the overall CMARL system behavior. Furthermore, it is important to verify how well defense methods defend the RL policies against adversarial attacks and therefore need to be integrated into the verification process too. Our method takes all of it into account. It takes advantage of the fact that CMARL agents have shared rewards and that all agent behaviors emerge together into a joint policy. These two CMARL properties allow us to verify CMARL agents by modeling the joint policy and environment as an induced DTMC. It is an efficient method because it only builds the model that is reachable via the trained policies and allows, therefore, the verification of larger models than what is possible via naive monolithic model checking.

This section is structured as follows. First, we explain how we model check deep CMARL agents. Second, we illustrate the attack setting, show how to attack a trained CMARL system, and how to model check such attacks. Third, we illustrate the defense setting and explain how to model check the denoisers in combination with the trained CMARL policies with and without adversarial attacks.

Model Checking of CMARL Agents

Recall, the joint policy π induced by the set of all agent policies $\{\pi_i\}_{i \in I}$ is a single policy π (Boutilier 1996). The tool COOL-MC[1] (Gross et al. 2022) allows model checking of a single RL policy against a user-provided PCTL property and MDP. Thereby, it builds the induced DTMC incrementally (Cassez et al. 2005).

To support joint policies, we created a *joint policy wrapper* that handles the generation of the observations for the RL policies π_i and builds the joint action $\pi(s)$ at every visited state s (see Figure 2). The wrapper takes (user-provided) observation functions $\mathbb{O}_i$ to map the states s to the corresponding observations o_i. Each of the CMARL policies π_i gets queried by its observation o_i to build the joint action $a = a_1 \times ... \times a_i$.

With the joint policy wrapper, we build the induced DTMC the following way. For every state s that is reachable via the joint policy π, we query for an action $a = \pi(s)$. In the underlying MDP M, only states s' that may be reached via that action $a \in A(s)$ are expanded. The resulting Markov chain induced by M and π is fully deterministic, as no action choices are left open, and ready for efficient model checking.

Our method is independent of the learning algorithm and allows the model checking of CMARL policies that select their actions based on current observations. Checking of probabilistic policies is supported by always choosing the action with the highest probability at every state. We support every environment that can be modeled via the PRISM language (Kwiatkowska, Norman, and Parker 2011).

[1]Download it from https://github.com/LAVA-LAB/COOL-MC. Our main features are supported.

Attack Setting

We now describe the adversarial attack setting (adversary's goals, knowledge, and capabilities).

Adversary's goal. The adversary aims to modify the performance of the trained CMARL agents in their environment. For instance, the adversary may try to increase the probability that the CMARL agent's production costs for a product exceed a threshold.

Adversary's knowledge. We consider an adversary that knows the weights θ_i of the trained policies (for the FGSM attack) and knows the CMMDP of the environment. Note that we can replace the FGSM attack with any other attack. Therefore, knowing the weights of the trained policies should not be a strict constraint.

Adversary's capabilities. Our attack setting allows the adversary to attack the trained policies at every visited state (see Figure 2c) during the incremental building process for the model checking of the adversarial-induced DTMC (offline) and after the RL policy got deployed (online).

Defense Setting

We now describe the defense setting (defender's goals, knowledge, and capabilities).

Defender's goal and knowledge. Our defense goal is to remove the adversarial attack from the agent's observations and retain the original observation (Serban and Poll 2018).

Defender's knowledge. The defender knows everything about the trained policy, the CMMDP of the environment, and the adversarial attack method.

Defender's capabilities. Our defense setting allows the defender to clean each observation of the trained policies during the incremental building process for the model checking (offline) and after the RL policy got deployed (online). A denoiser takes a clean observation (see Figure 2b) or an adversarial observation (see Figure 2d) as input (depending if the CMARL system is under attack) and outputs the cleaned observation.

Experiments

We now evaluate our CMARL model checking method in three benchmarks. First, we compare our CMARL model checking method with naive monolithic model checking and analyze the performance of the trained agents. Second, we analyze how many CMARL agents we can handle. Third, we analyze the agent performance change by using denoisers with and without adversarial attacks.

Setup

In this section, we explain the setup of our experiments [2] and detail the three case studies that we use for this paper. We applied our proposed method to an environment inspired by the *dynamic job shop scheduling problem (DJSSP)* from the IJCAI 2021 competition (Competitions 2021), an environment inspired by the *Flatland Challenge: Multi-Agent Reinforcement Learning on Trains (TRAINS)* from the ICAPS 2021

[2]Reproduce the experiments with the code from https://github.com/LAVA-LAB/CMARL-VERIFICATION

competition (Competitions 2021), and the dining philosopher QComp benchmark (Hartmanns et al. 2019) transformed into a CMARL system.

DJSSP. We transformed the single RL environment of the IJCAI 2021 challenge into a CMARL environment. Our CMARL environment comprises a manufacturing system with a set of jobs that must be manufactured via several autonomous machines before a given deadline (see Figure 3).

Stochastic events such as random machine breakdowns and changing production costs, all of which frequently happen in real-world manufacturing (Popper et al. 2021), are considered in this environment.

Each machine is controlled by a single agent i. We allow parallel working on multiple jobs simultaneously. However, if two machines work on the same job simultaneously, the environment terminates. Each job $j \in J$ consists of a sequence of operations; each should be processed on a specific machine i and takes a particular time, namely the processing time $T \in \mathbb{Z}^{ij}$. For example, $T_{13} = 2$ indicates that machine 3 has to execute its operation two time steps for job 1.

Each state s consists of features for the total number of processing times T_{ji} for each job j and machine i, a feature for the status of each machine $z_i \in I \cup \{0\}$ (agent i is working on the job j, agent i no operation), a feature about the current hour of the day $time \in \mathbb{Z}$, a feature about the available budget $budget \in \mathbb{Z}$, and a feature about the current energy price $price \in \mathbb{Z}$ (extracted from (Tveten, Bolkesjø, and Ilieva 2016)).

Each operation costs energy, and at every time step, the energy price may vary. If there is no more budget left, the environment terminates. Operations depend on each other. For instance, in our setting, operation 2 can not be executed before operation 1 and operation 3 have been done ($T_{i1} == 0 \wedge T_{i3} == 0$). If an operation is executed in the wrong order, the environment terminates. Uncertainty is introduced by the effect that operations may delay in 10% of cases (a machine breaks down for a time step).

The observation function $\mathbb{O}_i$ for agent i maps the current state s to the observation o_i. An observation o_i consists of the current energy price $price$, the hour of the day $time$, all processing times T_{ji} that must be done by agent i and all the processing times of operations on which it directly depends and the working status of the machines z_i.

Each agent i has a discrete action space A_i, which includes take-actions that let the agent choose a specific job to work on and an action for no-operation.

The CMARLS receive their cooperative penalty as soon as the environment terminates. If all jobs were finished, the penalty consists of the spent budget and the number of needed time steps to finish all jobs. Otherwise, it gets a penalty of 200 minus the number of executed operations.

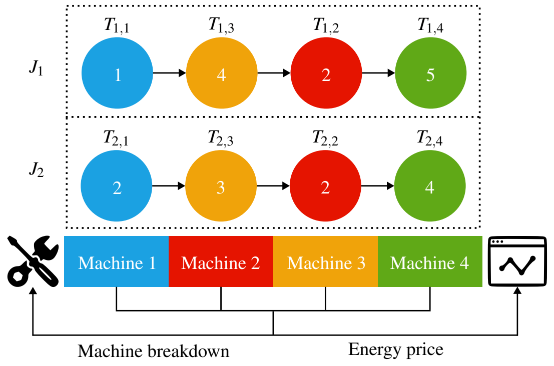


Figure 3: In the DJSSP environment, two jobs with different operation processing times need to be finished. Each machine is controlled by a CMARL agent and takes care of one specific operation. A machine can break down, and the agents know the energy price.

The DJSSP environment has $10,331,493$ states and $2,037,223,057$ transitions. *Differences to the IJCAI 2021 competition:* random machine breakdown is simulated in our case as no operation in the given time step, but the machine can be used in the next time step again; each machine is controlled by an agent; each agent partially observes the environment; and we added fluctuating energy prices and safety requirements like operation collisions and production cost thresholds.

$$
\begin{aligned}
S &= \{(T, price, budget, z_1, z_2, z_3, z_4), \dots\} \\
\mathbb{O}_1(s) &= \{(T_{11}, T_{21}, price, budget, time, z_1, z_2, z_3, z_4), \dots\} \\
\mathbb{O}_2(s) &= \{(T_{11}, T_{21}, T_{12}, T_{22}, T_{13}, T_{23}, price, budget, time, \\
&\qquad z_1, z_2, z_3, z_4), \dots\} \\
\mathbb{O}_3(s) &= \{(T_{13}, T_{23}, price, budget, time, z_1, z_2, z_3, z_4), \dots\} \\
\mathbb{O}_4(s) &= \{(T_{14}, T_{24}, price, budget, time, z_1, z_2, z_3, z_4), \dots\} \\
A_i &= \{NOP, take1, take2\} \text{ for each agent } i \\
A &= A_1 \times A_2 \times A_3 \times A_4
\end{aligned}
$$

$$
penalty = \begin{cases} \text{if operations are done,} \\ \text{time steps + initial budget - budget} \\ \text{else,} \\ 200 - \text{ number of executed operations} \end{cases}
$$

TRAINS. This environment consists of three trains that try to reach their destination (see Figure 4). The environment terminates as soon as they reach their destination, the time runs out, or there is a collision between the trains. The CMARL agents receive a cooperative penalty as soon as the environment terminates. It consists of 100 if there was a collision or time out or not all trains arrived in time. Otherwise, the penalty is 100 minus the number of arrived trains minus the remaining time. In 5% of cases, a train must stop because of malfunctioning, and the chosen action is not executed. The TRAIN environment has 217 states and $31,372$ transitions. *Differences to the ICAPS 2021 competition:* train collisions let the environment terminate; and all trains must reach their destination in the same time span.

Env.	Label	PCTL Property Query ($P(\phi)$)	=	$\|S\|$	$\|T\|$	Time (s)
DJSSP	done	$P(F\ jobs\_done)$	0.58	11629	27157	8440
DJSSP	collision	$P(F\ collision)$	0.22	11479	27007	6432
DJSSP	wrong_order	$P(F\ wrong\_order)$	0.25	11711	27239	8162
DJSSP	time	$P(F\ time)$	0	11871	27399	8335
DJSSP	bankruptcy	$P(F\ no\_budget)$	0.004	9241	24769	5065
TRAINS	arrived	$P(F\ arrived)$	0.99	11	16	7
TRAINS	delayed	$P(F\ delay)$	0.0975	10	15	6
TRAINS	crashed	$P(F\ crash)$	0	13	18	8
TRAINS	train21	$P(\neg$train1_arrives U train2_arrives$)$	0.99	8	11	4
DPP3	end	$P(F\ done)$	0	7	7	1

Table 1: PCTL property queries, with their labels and the original result of the property query without a denoiser and attack (=). The MDP of the DJSSP environment has $10,331,493$ states and $2,037,223,057$ transitions, the MDP of the TRAIN environment has 217 states and $31,372$ transitions, and the DPP3 MDP has 190 states and 855 transitions.

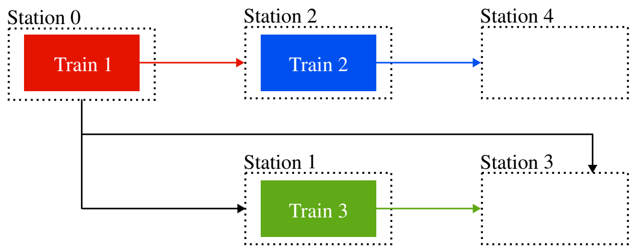


Figure 4: Train network of different train lines (different colors). For instance, the red train starts at station 0 and has to arrive at station 2.

$$S = \{(done, time, agent0\_id, agent0\_target\_id, agent0\_moving, ...), ...\}$$
$$\mathbb{O}_1(s) = \mathbb{O}_2(s) = \mathbb{O}_3(s) = S$$
$$A_i = \{nop, left, straight, right, stop\} \text{ for each agent } i$$
$$A = A_1 \times A_2 \times A_3$$
$$penalty = \begin{cases} \text{if collision or time out and not all trains arrived,} \\ 100 \\ \text{if no collision and no time out and trains arrived,} \\ 100 \text{ - number of arrived trains - remaining time} \end{cases}$$

Dining philosophers problem (DPPN). There are N agents seated around a circular table. To the left of each agent lays a fork, and in the center stands a bowl of spaghetti. An agent is expected to spend most of its time thinking; but when the agent feels hungry, it needs to pick up both the left and right fork to eat the spaghetti. When the agent finishes the meal, it puts down both forks and continues thinking. A fork can be used by only one agent at a time. The environment terminates when two agents try to grab the same fork (fork collision) or one of the agents starves. The agents must collaborate so that nobody starves and no fork collisions happen. For every additional time step the environment does not terminate, the agents get a reward.

$$S = \{(\text{hunger level 1, hunger level 2, ..., hunger level N}), ...\}$$
$$\mathbb{O}_i(s) = \text{hunger levels of neighbor agents and its own hunger level}$$
$$A_i = \{eat, think\} \text{ for each agent } i$$
$$A = A_1 \times A_2 \times ... \times A_i$$
$$reward = 1, \text{ for each time step}$$

Trained RL policies. We trained three CMARL agents in the environments such that they organized themselves to reach their goals. All CMARL agents were trained with separate deep Q-learning algorithms (Mnih et al. 2013) with a common reward function (Tampuu et al. 2015). We set for all training runs the Numpy random seed = 128, PyTorch random seed = 128, and Storm random seed = 128. We used $\varepsilon = 0.1$ ($\varepsilon = 0.5$ for the TRAIN agents), $\varepsilon_{decay} = 0.9999$, $\varepsilon_{min} = 0.01$ ($\varepsilon_{min} = 0.1$ for the TRAIN agents), $\gamma = 0.99$, a target network replacement of 304, batch size of 32, and a replay buffer size of $300,000$. Each neural network consists of two layers, each with 128 neurons (64 neurons for the TRAIN agents). The DJSSP CMARL training consisted of $32,462$ epochs with a best sliding window (window size = 100) reward of -60.98. The TRAIN CMARL training consisted of $25,000$ epochs with a best sliding window (window size = 100) reward of -96.24. The DPP3 training consisted of $10,000$ epochs with a best sliding window (windows size = 100) reward of 9.75.

Properties. Table 1 presents the performance of the policies for different properties (=). For instance, $done = 0.58$ describes the probability of the CMARL agents finishing all manufacturing jobs. Note at this point that we do not focus on achieving optimal performance but rather showing that it is possible to model check.

Technical setup. We executed our benchmarks on an NVIDIA GeForce GTX 1060 Mobile GPU, 16 GB RAM, and an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz x 12.

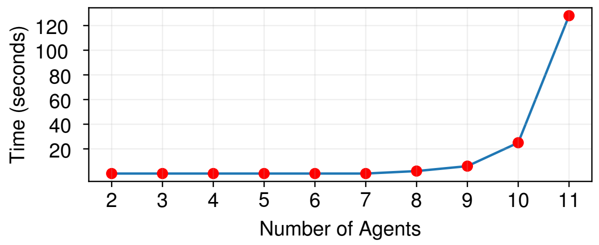


Figure 5: The exponential building time of each state for different numbers of CMARL agents via our method.

Analysis

We now answer the following research questions.

Can we model check CMARL environments that are too large for other model checkers? The naive monolithic model checking via Storm gives us the maximal reachability probability $P^{max}(\mathrm{F}\ arrived) = 0.99$ that all trains arrive at their destination. We model checked our trained CMARL agents via our method, and we observed that our agents achieve the same performance ($arrived = 0.99$).

For the DJSSP environment, it is intractable to check the MDP of the DJSSP (10,331,493 states and 2,037,223,057 transitions) via naive monolithic model checking. Storm runs out of memory after 22 minutes with the property query $P^{max}(\mathrm{F}\ jobs\_done)$. Our method, on the other hand, gives us a reachability probability for $done = 0.58$ (see Table 1). However, at some point, our model checking method is also limited by the size of the induced DTMC and runs out of memory (Gross et al. 2022).

How many agents can our method handle? We now analyze how many agents our CMARL model checking method can handle. In this experiment, we focus on the DPPN environment because it is straightforward to scale. We train CMARL agents in different DPPN environments with different numbers of agents. Our experiment shows, that we can handle up to 18 agents at the same time. At 19 agents, the model becomes too large to parse. At every incremental building process step, a callback function has to be called $|Act(s)|^{|I|} = 2^{|I|}$ times per state. Therefore, the model building time becomes expensive (you can track the building times for each state for different numbers of agents in Figure 5). With our technical setup, the naive monolithic model checking takes around $1 \cdot 10^{-5}$ seconds for each state (independent of the number of agents). During the model checking of 11 agents and the property *end*, the naive monolithic model checking runs out of memory while our method still allows the model checking of them. We conclude that the model checking of CMARL systems is also limited by the number of agents.

How do adversarial attacks influence the performance of the trained CMARL agents? We now analytically measure the impact of adversarial attacks in our environments. Therefore, we create at every visited state an α-bounded FGSM attack ($\alpha = 0.1$) for each policy π_i during the incremental building process of the induced DTMC (see an example for an attack in Figure 2c). Our experiments show that FGSM attacks influence the performance of the CMARL system (compare column $=$ and $=_{adv}$ in Table 2).

Label	$=$	$=_{adv}$	$=^{denoiser}$	$=^{denoiser}_{adv}$
¬done	0.42	0.44	0.42	0.42
collision	0.22	0.26	0.22	0.22
wrong_order	0.25	0.25	0.25	0.25
time	0	0	0	0
bankruptcy	0.004	0.1	0.004	0.004
¬arrived	0.01	0.1	0.01	0.01
crashed	0	0	0	0
delayed	0.0975	0.0975	0.0975	0.0975
train21	0.99	0.99	0.99	0.99
end	0	1	1	1

Table 2: Comparison between no denoiser and no attack ($=$), no denoiser and attack ($=_{adv}$), denoiser and no attack ($=^{denoiser}$), and denoiser and attack ($=^{denoiser}_{adv}$). All attacks are bounded by $\alpha = 0.1$.

How does the CMARL agents' performance change by equipping them with denoisers? We now analytically measure how well CMARL policies perform with denoisers under and not under attack. The adversary and defender operate before the observations get passed to the agents (see Figure 2d). For each agent, there is a separate adversary attack and a separate denoiser. The adversarial attacks are created via FGSM. We trained each denoiser the following way: 1. During CMARL policy training, we collected k states Y ($k = 1000$ for DJSSP, $k = 11$ for TRAINS, $k = 7$ for DPP3). 2. For each $y \in Y$, we create an adversarial state x via the FGSM attack ($\alpha = 0.1$) and store the data point (x, y) and (y, y) into the data set A. 3. We added synthetic data (plus m data points) to the dataset by randomly shuffling the vector elements of each x ($m = 19000$ for DJSSP, $m = 9890$ for TRAINS, $m = 9993$ for DPP3). 4. Train denoiser on A for 100 episodes with $seed = 860, 523, 297, 119, 962, 652$, learning rate $= 0.0001$, batch size $= 32$, and four neural network layers (each with 1048 neurons). The losses of the denoisers vary between 0.0005 and $0.2 \cdot 10^7$.

Table 2 shows in column $=^{denoiser}$ that, in most cases, the denoisers do not decrease the performance of the policies. In column $=^{denoiser}_{adv}$, we observe that, in most cases, the denoisers under α-bounded FGSM attacks ($\alpha = 0.1$) also do not decrease the performance of the policies.

Conclusion

Our method checks trained CMARL agents equipped with or without denoisers in adversarial or non-adversarial environments to ensure compliance with safety requirements after deployment. It has been successfully applied to real-life applications such as job scheduling, transportation, and resource allocation. However, the size of the induced DTMC and the number of CMARL agents limit our method. Optimizing the incremental model building process of COOL-MC can increase the number of supported CMARL agents. Incorporating safe CMARL approaches would also be valuable extensions to our method, as already done in the single RL domain (Carr et al. 2023; Jin et al. 2022; Jothimurugan et al. 2022; Jansen et al. 2020).

Acknowledgements

The research leading to this work was partially funded by the Belgian FWO G030020N project "SAILor". Furthermore, this work was partially funded by the SAM-FMS project (NWO project 17931) in the MasCot program, and it was supported by the ERC Starting Grant 101077178 (DEUCE).

References

Amodei, D.; Olah, C.; Steinhardt, J.; Christiano, P. F.; Schulman, J.; and Mané, D. 2016. Concrete Problems in AI Safety. *CoRR*, abs/1606.06565.

Baier, C.; and Katoen, J. 2008. *Principles of model checking*. MIT Press.

Bakhti, Y.; Fezza, S. A.; Hamidouche, W.; and Déforges, O. 2019. DDSA: A Defense Against Adversarial Attacks Using Deep Denoising Sparse Autoencoder. *IEEE Access*, 7: 160397–160407.

Bengio, Y.; Yao, L.; Alain, G.; and Vincent, P. 2013. Generalized Denoising Auto-Encoders as Generative Models. In *NIPS*, 899–907.

Bertrand, N.; and Fournier, P. 2013. Parameterized Verification of Many Identical Probabilistic Timed Processes. In *FSTTCS*, volume 24 of *LIPIcs*, 501–513. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Boutilier, C. 1996. Planning, Learning and Coordination in Multiagent Decision Processes. In *TARK*, 195–210. Morgan Kaufmann.

Carr, S.; Jansen, N.; Junges, S.; and Topcu, U. 2023. Safe Reinforcement Learning via Shielding under Partial Observability. In ***AAAI***.

Cassez, F.; David, A.; Fleury, E.; Larsen, K. G.; and Lime, D. 2005. Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In *CONCUR*, volume 3653 of *LNCS*, 66–80. Springer.

Chen, T.; Kwiatkowska, M. Z.; Parker, D.; and Simaitis, A. 2011. Verifying Team Formation Protocols with Probabilistic Model Checking. In *CLIMA*, volume 6814 of *LNCS*, 190–207. Springer.

Competitions. 2021. Competitions. https://icaps21.icaps-conference.org/Competitions/. [Online; accessed 01-11-2022].

Courcoubetis, C.; and Yannakakis, M. 1988. Verifying Temporal Properties of Finite-State Probabilistic Programs. In *FOCS*, 338–345. IEEE Computer Society.

Courcoubetis, C.; and Yannakakis, M. 1995. The Complexity of Probabilistic Verification. *J. ACM*, 42(4): 857–907.

Dablain, K. 2017. *Cyber threats against critical infrastructures in railroads*. Ph.D. thesis, Utica College.

Fujimoto, T.; and Pedersen, A. P. 2021. Adversarial Attacks in Cooperative AI. *CoRR*, abs/2111.14833.

Gross, D.; Jansen, N.; Junges, S.; and Pérez, G. A. 2022. COOL-MC: A Comprehensive Tool for Reinforcement Learning and Model Checking. In *SETTA*. Springer.

Hahn, E. M.; Perez, M.; Schewe, S.; Somenzi, F.; Trivedi, A.; and Wojtczak, D. 2019. Omega-Regular Objectives in Model-Free Reinforcement Learning. In *TACAS (1)*, volume 11427 of *LNCS*, 395–412. Springer.

Hansson, H.; and Jonsson, B. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects Comput.*, 6(5): 512–535.

Hartmanns, A.; Klauck, M.; Parker, D.; Quatmann, T.; and Ruijters, E. 2019. The Quantitative Verification Benchmark Set. In *TACAS (1)*, volume 11427 of *LNCS*, 344–350. Springer.

Hasanbeig, M.; Kroening, D.; and Abate, A. 2020. Deep Reinforcement Learning with Temporal Logics. In *FORMATS*, volume 12288 of *LNCS*, 1–22. Springer.

Hensel, C.; Junges, S.; Katoen, J.; Quatmann, T.; and Volk, M. 2022. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4): 589–610.

Huang, S. H.; Papernot, N.; Goodfellow, I. J.; Duan, Y.; and Abbeel, P. 2017. Adversarial Attacks on Neural Network Policies. In *ICLR (Workshop)*. OpenReview.net.

Hüttenrauch, M.; Sosic, A.; and Neumann, G. 2019. Deep Reinforcement Learning for Swarm Systems. *J. Mach. Learn. Res.*, 20: 54:1–54:31.

Ilahi, I.; Usama, M.; Qadir, J.; Janjua, M. U.; Al-Fuqaha, A. I.; Hoang, D. T.; and Niyato, D. 2022. Challenges and Countermeasures for Adversarial Attacks on Deep Reinforcement Learning. *IEEE Trans. Artif. Intell.*, 3(2): 90–109.

Im, D. J.; Ahn, S.; Memisevic, R.; and Bengio, Y. 2017. Denoising Criterion for Variational Auto-Encoding Framework. In *AAAI*, 2059–2065. AAAI Press.

Jansen, N.; Könighofer, B.; Junges, S.; Serban, A.; and Bloem, R. 2020. Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper). In *CONCUR*, volume 171 of *LIPIcs*, 3:1–3:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Jin, P.; Tian, J.; Zhi, D.; Wen, X.; and Zhang, M. 2022. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In *CAV (1)*, volume 13371 of *LNCS*, 193–218. Springer.

Jothimurugan, K.; Bansal, S.; Bastani, O.; and Alur, R. 2022. Specification-Guided Learning of Nash Equilibria with High Social Welfare. In *CAV (2)*, volume 13372 of *LNCS*, 343–363. Springer.

Junges, S.; Jansen, N.; Katoen, J.; Topcu, U.; Zhang, R.; and Hayhoe, M. M. 2018. Model Checking for Safe Navigation Among Humans. In *QEST*, volume 11024 of *LNCS*, 207–222. Springer.

Khan, A.; Zhang, C.; Li, S.; Wu, J.; Schlotfeldt, B.; Tang, S. Y.; Ribeiro, A.; Bastani, O.; and Kumar, V. 2019. Learning Safe Unlabeled Multi-Robot Planning with Motion Constraints. In *IROS*, 7558–7565. IEEE.

Kwiatkowska, M.; Norman, G.; and Parker, D. 2019. Verification and Control of Turn-Based Probabilistic Real-Time Games. In *The Art of Modelling Computational Systems*, volume 11760 of *LNCS*, 379–396. Springer.

Kwiatkowska, M.; Norman, G.; Parker, D.; and Santos, G. 2020. Multi-player Equilibria Verification for Concurrent Stochastic Games. In *QEST*, volume 12289 of *LNCS*, 74–95. Springer.

Kwiatkowska, M.; Norman, G.; Parker, D.; and Santos, G. 2021. Automatic verification of concurrent stochastic systems. *Formal Methods Syst. Des.*, 58(1-2): 188–250.

Kwiatkowska, M. Z.; Norman, G.; and Parker, D. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*, volume 6806 of *LNCS*, 585–591. Springer.

Lin, J.; Dzeparoska, K.; Zhang, S. Q.; Leon-Garcia, A.; and Papernot, N. 2020. On the Robustness of Cooperative Multi-Agent Reinforcement Learning. In *SP Workshops*, 62–68. IEEE.

Lomuscio, A.; and Pirovano, E. 2020. Parameterised Verification of Strategic Properties in Probabilistic Multi-Agent Systems. In *AAMAS*, 762–770. International Foundation for Autonomous Agents and Multiagent Systems.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602.

Moos, J.; Hansel, K.; Abdulsamad, H.; Stark, S.; Clever, D.; and Peters, J. 2022. Robust Reinforcement Learning: A Review of Foundations and Recent Advances. *Mach. Learn. Knowl. Extr.*, 4(1): 276–315.

Mqirmi, P. E.; Belardinelli, F.; and León, B. G. 2021. An Abstraction-based Method to Check Multi-Agent Deep Reinforcement-Learning Behaviors. In *AAMAS*, 474–482. ACM.

Ohashi, K.; Nakanishi, K.; Sasaki, W.; Yasui, Y.; and Ishii, S. 2021. Deep Adversarial Reinforcement Learning With Noise Compensation by Autoencoder. *IEEE Access*, 9: 143901–143912.

Popper, J.; Motsch, W.; David, A.; Petzsche, T.; and Ruskowski, M. 2021. Utilizing Multi-Agent Deep Reinforcement Learning For Flexible Job Shop Scheduling Under Sustainable Viewpoints. In *ICECCME*, 1–6. IEEE.

Qin, W.; Sun, Y.-N.; Zhuang, Z.-L.; Lu, Z.-Y.; and Zhou, Y.-M. 2021. Multi-agent reinforcement learning-based dynamic task assignment for vehicles in urban transportation system. *International Journal of Production Economics*, 240: 108251.

Riley, J.; Calinescu, R.; Paterson, C.; Kudenko, D.; and Banks, A. 2021a. Reinforcement Learning with Quantitative Verification for Assured Multi-Agent Policies. In *ICAART (2)*, 237–245. SCITEPRESS.

Riley, J.; Calinescu, R.; Paterson, C.; Kudenko, D.; and Banks, A. 2021b. Utilising Assured Multi-Agent Reinforcement Learning within Safety-Critical Scenarios. In *KES*, volume 192 of *Procedia Computer Science*, 1061–1070. Elsevier.

Schuppe, G. F.; and Tumova, J. 2021. Decentralized Multi-Agent Strategy Synthesis under LTLf Specifications via Exchange of Least-Limiting Advisers. In *MRS*, 119–127. IEEE.

Serban, A. C.; and Poll, E. 2018. Adversarial Examples - A Complete Characterisation of the Phenomenon. *CoRR*, abs/1810.01185.

Serrano-Ruiz, J. C.; Mula, J.; and Poler, R. 2021. Smart manufacturing scheduling: A literature review. *Journal of Manufacturing Systems*, 61: 265–287.

Shalev-Shwartz, S.; Shammah, S.; and Shashua, A. 2016. Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. *CoRR*, abs/1610.03295.

Tampuu, A.; Matiisen, T.; Kodelja, D.; Kuzovkin, I.; Korjus, K.; Aru, J.; Aru, J.; and Vicente, R. 2015. Multiagent Cooperation and Competition with Deep Reinforcement Learning. *CoRR*, abs/1511.08779.

Tveten, Å. G.; Bolkesjø, T. F.; and Ilieva, I. 2016. Increased demand-side flexibility: market effects and impacts on variable renewable energy integration. *International Journal of Sustainable Energy Planning and Management*, 11: 33–50.

Vamplew, P.; Smith, B. J.; Källström, J.; de Oliveira Ramos, G.; Radulescu, R.; Roijers, D. M.; Hayes, C. F.; Heintz, F.; Mannion, P.; Libin, P. J. K.; Dazeley, R.; and Foale, C. 2022. Scalar reward is not enough: a response to Silver, Singh, Precup and Sutton (2021). *Auton. Agents Multi Agent Syst.*, 36(2): 41.

Vincent, P.; Larochelle, H.; Bengio, Y.; and Manzagol, P. 2008. Extracting and composing robust features with denoising autoencoders. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, 1096–1103. ACM.

Wang, L.; Ames, A. D.; and Egerstedt, M. 2016. Safety barrier certificates for heterogeneous multi-robot systems. In *ACC*, 5213–5218. IEEE.

Wang, Y.; Roohi, N.; West, M.; Viswanathan, M.; and Dullerud, G. E. 2020. Statistically Model Checking PCTL Specifications on Markov Decision Processes via Reinforcement Learning. In *CDC*, 1392–1397. IEEE.

Wong, A.; Bäck, T.; Kononova, A. V.; and Plaat, A. 2021. Multiagent Deep Reinforcement Learning: Challenges and Directions Towards Human-Like Approaches. *CoRR*, abs/2106.15691.

Zhang, Y.; Zhu, H.; Tang, D.; Zhou, T.; and Gui, Y. 2022. Dynamic job shop scheduling based on deep reinforcement learning for multi-agent manufacturing systems. *Robotics Comput. Integr. Manuf.*, 78: 102412.

Zong, K.; and Luo, C. 2022. Reinforcement learning based framework for COVID-19 resource allocation. *Computers & Industrial Engineering*, 167: 107960.