

Integrating Distributed Component-Based Systems through Deep Reinforcement Learning*


Itay Cohen and Doron Peled

Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Modern system design and development often consists of combining different components developed by separate vendors under some known constraints that allow them to operate together. Such a system may further benefit from further refinement when the components are integrated together. We suggest a learning-open architecture that employs deep reinforcement learning performed under weak assumptions. The components are “black boxes”, where their internal structure is not known, and the learning is performed in a distributed way, where each process is aware only on its local execution information and the global utility value of the system, calculated after complete executions. We employ the proximal policy optimization (PPO) as our learning architecture adapted to our case of training control for black box components. We start by applying the PPO architecture to a simplified case, where we need to train a single component that is connected to a black box environment; we show a stark improvement when compared to a previous attempt. Then we move to study the case of multiple components.

1 Introduction

The process of system development often involves partitioning the system’s task into different components that work in tandem. These components can be developed in-house by different groups and can also include parts outsourced to other software houses or off-the-shelf elements. Recently, we see a growing research effort on the automatic construction of system components directly from the specification. We are also starting to witness the automatic synthesis of code based on natural language requirements, e.g., as part of the capabilities of the chatbot ChatGPT [1] that is based on large-scale deep learning. Constructing a system from concurrently performing components that are developed by different groups and using different methodologies may require some means of control for optimizing the combined behavior. In particular, the goal of such control can be the following: as components can interact with one another, one would like to avoid situation where a large number of attempts to interact would be unsuccessful due to the lack of knowledge of components about the current state of one another.

*  This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 956123 - FOCETA.

In this work, we address the problem of providing distributed control to a system that is developed from multiple components that interact with each other, i.e., a *multi-agent system*. The internal structure of these components is unknown (due to their development process), i.e., each component is considered to be a *black box* with respect to each other; only the types of interactions with the other processes is known to each process. We suggest that the components are designed with local provisions for establishing control based on deep learning. We do not want to rely on shared information for achieving control during their joint execution. Thus, the learning process can use only the locally observed information of a component.¹ Our goal is then to choose an effective learning-open architecture for distributed control for optimizing the joint execution of a multi-agent system. As a specific goal that we can demonstrate and experiment with, we chose to minimize the overall number of failed interactions between the processes (components) of a multi-agent system, i.e., the number of times that a process offers an interaction to another process while the other process is not ready for it. While this optimization criterion is natural and attainable, our control synthesis method does not have to be confined to it. A more general goal we considered is to maximize the cumulative rewards assigned for each type of action in a process (component).

Since each component sees the rest of the system as a black box with which it can interact, we adopt a reinforcement learning approach where the control is synthesized based on observations performed on the integrated system. However, due to using components developed independently, during training each component has access only to its local information. Our suggested approach can be considered as a version of *multiagent reinforcement learning (MARL)* [2]. In particular, we employ *deep* multiagent reinforcement learning [9], where trained neural networks undertake the task of controlling the components. While there is a large number of models and architectures designed for MARL, we seek an architecture and a training methodology that best suite our targeted systems, consisting of concurrent black box components that interact with one another. We refrain from allowing interactions between the learning components, e.g., the sharing of information at training time (as of a shared critique) [15]. We also concentrate on using state-of-the art techniques, including the *proximal policy optimization* (PPO) algorithm [13], a reinforcement learning algorithm that is employed also in chatGPT.

We developed our approach in two steps. In the first step we considered a simplified model that has concurrent synchronous components, which was first introduced in [7]. This allowed us to concentrate first on the selection of the deep learning architecture and parameters, and to compare to experiments that were done with respect to this limited model. This model includes a finite state com-

¹ This can also be easily generalized to employ a utility function that is calculated from the local utility functions of each component; for example, each component can calculate some measure of success (e.g., in form of a discounting sum) on performing interactions, and the global utility can be the sum of this measure over the participating processes.

ponent that we call the *system*, which interacts with a black-box *environment*. The control is only imposed on the system component and the environment component is uncontrollable. In each step, the system and the environment stay synchronized, where the communication between them is in the form of a handshake. The system makes a choice for the next action, and the environment must follow that choice if the action is enabled. Otherwise, a failed interaction occurs and the system does not move while the environment makes some independent progress. The control enforces the choice of the system’s next action based on the available partial information, which involves its sequence of local states and actions that occurred so far and the indication of success/failure to interact with the environment at each point. The control goal is to minimize the number of times that the system will offer an action that will result in a failed interaction.

As part of the first step of our approach, we devise an alternative deep reinforcement learning approach for the simplified setting, based on the Proximal Policy Optimization algorithm. We integrate a Recurrent Neural Network (RNN) with the network architecture of PPO, to capture the long term history of the executions. We show that this approach improves the experimental results of [7].

After experimenting with the restricted model, the second step included a full and more realistic model that encapsulates multiple concurrent components executing asynchronously. Although the model is asynchronous, it preserves the handshake-like form of communication; a pair of components should offer each other the same type of action to establish a successful interaction. We introduce a generalized approach that derives control for *multiple* components that can interact with one another, where a separate control, based on a trained RNN, is imposed on each component. Controlling the different components is obtained in a distributed fashion, where each component considers the others as its environment. The only signal that is shared between the components is whether an interaction succeeded, failed or missed.

To motivate our deep learning framework, consider the dining philosophers problem, which is a classical problem involving scheduling and interaction between concurrent processes. It involves a number of philosophers (often five) sitting around a table, with a fork between each two of them. Each philosopher needs to capture two forks to eat. After eating, he can release the forks. In order to let all the philosophers eat, a strategy involving capturing the forks is needed. A bad strategy can, e.g., let each philosopher capture the left fork first; then they discover that no one can eat, then trying the right fork, etc. In [10], it was proven that there is no deterministic (i.e., non probabilistic) and symmetric solution to the dining philosophers problem. In our framework, there is also the additional factor that the philosophers do not know the structure of each other (they are black boxes), hence the sought strategy needs to be discovered by experiments. Our deep reinforcement learning approach learns such strategies, represented within the used neural networks.

We implemented our proposed methodology using the PyTorch library [12]. To test our implementation, we devised small but somewhat challenging examples. These examples allowed us to evaluate the results of our experiments. The

different parameters of our approach were also fine-tuned as a result of experimenting with these examples. Our experiments consist of a comparison between the performance of our implementation and the optimal strategy for each example.

2 Preliminaries

We study systems that are constructed from finite state components and interact with each other. At any state a component can choose between a set of actions. A controller supervising a component can observe the current state of the system and can impose restrictions to the set of actions allowed at the current state. Such a mechanism is standard in control theory, and a formalization can be found e.g., in [3]. In particular, we look at a system constructed from multiple components running concurrently. The actions that are mutual to a pair of components can be considered as *interactions* and both components need to select the same action in order to successfully interact. In models like CSP [6] the interaction can be asymmetric, where one side is a sender and the other one is a receiver. In other models the interactions are defined in a more general context, not necessarily associated with message passing [4].

A Simplified Model Initially, we look at a simplified case, where one component (process) is a given state machine that we term “the system”, and the other one is a black box machine we term “the environment”. The set of actions is joined by both components. We further assume that the system and the environment are executing synchronously. The system has priority in selecting the next action (it is the “master”) and if the action it offers the environment is currently available by the environment (which is the “slave”), they will both interact and move to their respective state. Note that the state of the environment and the set of actions enabled from it are unknown, as the environment is a black box. If the environment cannot participate in the action offered to it by the system (it is not enabled), then it chooses one of its other enabled actions and progresses according to it. Note that here, for simplicity of the model, the environment progresses on one of the interactions without the system taking part in it; this modeling assumption will be removed in the more advanced model. In the simplified model case, the control is enforced only on the system, where the environment is to be left as is.

The details of this model were selected in order to simplify both the implementation and the presentation. This is also the model used in [7] and adopting the model allows us to compare the training architecture that we use here with the results of that work.

Example Consider the system in Figure 1 (left) and its environment (right). This system can always make a choice between the actions a, b and c , and the environment has to agree with that choice if it is enabled from its current state.

Remember that the system is unaware of the environment’s internal state. If the system selects actions according to $(abc)^*$ then the environment can follow that selection with no failures. On the other hand, if the system selects actions according to $(baa)^*$ it will fail constantly, while the environment keeps changing states. Our goal is to construct a control that restricts the system’s actions at each step such that the number of failed interactions will be minimal.

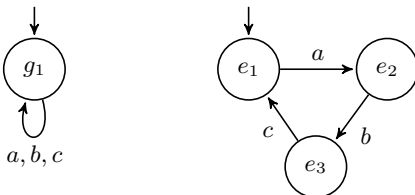


Fig. 1: *permitted*: System (left) and Environment (right).

A More Advanced Model In the advanced model, components (processes) can interact with one another. All the interactions involve pairs of components. Each component may also have local actions (which may model interactions with yet another component, when we want to abstract away some of the components). All the components are black boxes. There is no explicit notion of *environment* in this model; in a sense, each process can consider the rest of the processes combined as its environment. Control is applied, separately, to each component. If an agreed upon interaction is not selected, a component can select to perform a local action, but unlike the simpler case, cannot decide to participate unilaterally in an interaction. The execution is now asynchronous, with components performing according to their own speeds. The choice of an asynchronous execution model further justifies the use of deep reinforcement learning as a method for learning to control the components; the relative speed of the different concurrent components contributes to the nondeterminism in the execution. Changing only the timing parameters of a single system would potentially change the optimal control strategies for the rest of the components. Hence it is the actual experience that is used in the training that allows us to effectively control the system; thus, a preliminary learning of finite state models of the components would take into account the timing parameters.

Reinforcement Learning Reinforcement learning (RL) includes methods for controlling the interaction with an environment [14]. The goal is to maximize the expected utility value that sums up the future rewards/penalties; these can be discounted by γ^n with respect to its future distance n from the current point, with $0 < \gamma \leq 1$, or be summed up with respect to a finite distance (horizon). Typically, the model for RL is a Markov Decision Process (MDP), where there is a probability distribution on the states that are reached by taking an action from

a given state. When the current state of the environment is not directly known to the controller during the execution, the model is a Partially Observable MDP (POMDP).

A *value-based* control policy (strategy) can be calculated by maximizing either a state-value function $V(s)$ or a state-action value $Q(s, a)$. Assuming acting according to a certain strategy, the value function $V(s)$ is equal to the expected discounted sum of rewards of a controller starting from state s . The state-value function means the expected discounted sum of rewards received by the controller starting with an action a from state s . When the environment is fully observable, and the model that indicates the probability to move from one state to another (given an action) is known, a procedure based on Bellman's equation [14] can be used. If the probabilities are unknown but we can still observe the environment, a randomized-based (*Monte Carlo*) exploration method can be used to update the value function and converge towards the optimal policy.

Policy based RL methods avoid calculating the optimal utility value directly at each state, hence are more effective when the number of possible states is huge. The policy is parametric and its parameters are optimized based on gradient descent. Such parameters can be, in particular, the weights of a neural network. The class of reinforcement learning algorithms that utilize neural networks to represent a parametric policy, or to estimate state-action values is called *deep reinforcement learning*.

Deep Learning Deep learning is a collection of methods for training *neural networks*, which can be used to perform various tasks such as image and speech recognition or playing games at an expert level. A neural network consists of a collection of nodes, the *neurons*, arranged in several layers, where each neuron is connected to all the neurons in the previous and the next layer. The first layer is the *input layer* and the last layer is the *output layer*. The other layers are *hidden*.

The value x_i of the i^{th} neuron at layer $j + 1$ is computed from the column vector $\mathbf{y} = (y_1, \dots, y_m)$ of all the neurons at layer j . To compute x_i , we first apply a transformation $t_i = \mathbf{w}_i \mathbf{y} + b_i$ where \mathbf{w}_i is a line vector of *weights*, and b_i a number called the *bias*. Then we apply to the vector $\mathbf{t} = (t_1, \dots, t_n)$ an *activation function*, making the value of each neuron a non-linear function of the values of neurons at the preceding layer. Typical activation functions include the *sigmoid* and *tanh* functions, as well as *ReLU* and *softmax*.

The ReLU (rectified linear unit) activation function is defined as the positive part of its argument. When applied on a vector, it is defined as a vector of the positive part of each of its coordinates.

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The *softmax* activation function takes a vector of values and normalizes it into a corresponding vector of probability distributions, i.e., with values between

0 and 1, summing up to 1.

$$\text{softmax}(t_1, \dots, t_n) = \left(\frac{e^{t_1}}{\sum_i e^{t_i}}, \dots, \frac{e^{t_n}}{\sum_i e^{t_i}} \right)$$

Given values for all neurons in the input layer, we can compute the values for all neurons in the network. Overall, a neural network represents a function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ where n is the size of the input layer, and m the size of the output layer.

The values of the weights w_i and the biases b_i are modified through *training*. A *loss function* provides a measurement of the distance between the actual output of the neural network and the desired output. The goal of training is to minimize the loss function. Optimizing the parameters is performed from the output layer backwards based on gradient descent.

For applications where sequences of inputs are analyzed, as e.g., in language recognition, one often uses a form of network called *Recurrent Neural Network* (RNN). An RNN maintains a feedback loop, where values of some neurons are fed back to the network as additional inputs in the next step. In this way an RNN has the capability of maintaining some long term memory that summarizes the input sequence so far. A more specific type of RNN that intends to solve this problem is a *Long Short-Term Memory*, LSTM. It includes components that control what (and how much) is erased from the memory layer of the network and what is added.

Proximal Policy Optimization Proximal Policy Optimization (PPO) [13] is a deep reinforcement learning algorithm that is based on a policy gradient method; it searches the space of policies rather than assigning values to state-action pairs. The policy π is represented by a neural network, and its set of parameters is denoted by θ . A detailed description of the network architecture will follow. This algorithm uses the notion of the *advantage* of an action with respect to a certain state. An advantage $A(s, a)$ measures how much an action is a good or a bad decision given a certain state. Formally, is it defined as $A(s, a) = Q(s, a) - V(s)$. To trace the impact of the different actions, the algorithm calculates the probability of the action under the current policy $\pi(a | s)$, and divides it by the probability of the action under the previous policy $\pi_{old}(a | s)$. For timestep t , this ratio is denoted by $r_t(\theta)$, where

$$r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}$$

When $r_t(\theta)$ is greater than one, the relevant action is more probable in the current policy than in the old one. $r_t(\theta)$ that is between zero and one indicates that the action is less probable in the current policy than in the old one.

The algorithm uses the *clipped surrogate objective* function L^{CLIP} to optimize the policy's set of parameters θ . This function is maximized using stochastic gradient ascent. It is defined as follows:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)]$$

where

- \hat{E}_t denotes the empirical expectation over timesteps.
- \hat{A}_t is the estimated advantage of the selected action at time t .
- clip is a function that clips the values of $r_t(\theta)$ between $(1 - \epsilon, 1 + \epsilon)$. Formally, it is defined as follows:

$$\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) = \begin{cases} 1 + \epsilon & \text{if } r_t(\theta) \geq 1 + \epsilon \\ 1 - \epsilon & \text{if } r_t(\theta) \leq 1 - \epsilon \\ r_t(\theta) & \text{otherwise.} \end{cases}$$

Note that when $r_t(\theta)$ is not in the interval $(1 - \epsilon, 1 + \epsilon)$, the gradient of $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$ is zero.

The main idea of this objective function is to incentivize actions that led to higher rewards. However, we would like to restrict the amount that the policy can change in every optimization, to help guarantee that it is monotonically improving. This restriction is reflected by the clipping mechanism of the objective function. Consider a case where a certain action became significantly more probable under the current policy rather than the old one, and its advantage is positive, meaning that it had an estimated positive effect on the outcome. If there were no restrictions, we might perform a relatively big optimization step, that might destruct our policy in the future. With the clipping mechanism of $r_t(\theta)$, its gradient would be considered as zero if it exceeded the threshold of $1 + \epsilon$, and we would ignore the potential optimization step as a result.

According to the algorithm, we use the current policy π_θ to sample a constant number of episodes, and calculate the *reward-to-go* for every action in each episode. The reward-to-go of a state-action pair is the discounted sum of rewards from this point and up until the end of the episode. It is denoted by $\hat{R}(s_t, a_t)$. This value is often used as a local approximation of the state-action value $Q(s, a)$. The estimated advantage of a chosen action is then calculated by $\hat{A}_t = \hat{R}(s_t, a_t) - V_\varphi(s_t)$, where V_φ is a state-value function that is also optimized throughout the same learning process. Then, the policy parameters are optimized according to the clipped surrogate objective function. Later, the parameters of the state-value function are optimized according to the mean-squared loss with respect to the reward-to-go:

$$MSE(\varphi) = \hat{E}_t[(V_\varphi(s_t) - \hat{R}(s_t, a_t))^2]$$

3 The Proposed Approach

3.1 Overview

Initially, we suggest a PPO based approach that constructs a controller for the single system component in the simplified model. It strives to minimize the number of failures of interactions offered by the system component to the black box

environment component. Controlling the selection of the system component’s actions at each step assumes that it is unaware of the environment’s internal state. In fact, the only information that is available from the environment at each execution step is whether the interaction succeeded or not.

Later, in the case of the more advanced model which includes multiple components and local actions, we suggest an architecture where each component is learning-enabled, equipped with its own neural network. The neural network is trained to control the process within the context of the entire system, based on a given utility measurement. The training of the different components is performed simultaneously, based on executions of the entire system; however, each component is trained within that context locally, based on the local information of the components.

To get an informed decision regarding the next action to be selected by the system component, our proposed controller, utilizes both long term and short term histories of the current execution. The short term history consists of the latest execution step from the component’s perspective: the latest action selected by the component, whether this interaction was successful, and the current component state as a result of the interaction. The long term history is a finite representation of the past selected actions of the component, including their status (successful or not), and the past states. This finite representation is a summary of the execution so far from the component point of view. It is necessary to represent the long term history in a finite way, since we design our controller to operate under unbounded execution lengths.

At each execution step the controller receives both the short term and the long term histories, selects the next interaction to be offered, and updates the long term history representation accordingly.

3.2 Suggested Deep Learning Architecture

The Controller’s State Space Representation As mentioned above, the controller receives two inputs - the short and the long term histories. A short term history is represented by a matrix M of size $|S^s| \times |T^s|$, where S^s is the system’s state space and T^s is the system’s set of actions. We assume a fixed order on the elements in S^s and T^s . The only non-zero cell of the matrix is M_{ij} , if the last offered action was $t_j \in T^s$, and the current system state as a result is $s_i \in S^s$. The value of M_{ij} is 1 if the interaction was successful and -1 otherwise. The controller’s state space is the set of all valid short term histories, denoted by S' . Note that the state space does not provide any information about the long term history. This is due to the fact that short term history usually have greater impact on the selection of future actions than long term history. Hence, we wanted to emphasize these elements in terms of the actor network’s course of action.

Actor-Critic based Implementation To construct the controller, we implemented a variant of the PPO algorithm with a few modifications. Our variant relied on the *Actor-Critic* architecture [5].

Actor-Critic learning is a deep reinforcement learning technique that utilizes both an actor neural network and a critic neural network. The actor network is responsible for generating *actions* based on the current state, while the critic network evaluates the quality of the different *states* of the network. The two networks are trained in a cooperative manner, where the idea is that the critic provides feedback to the actor. To capture the long term history in a finite representation, we incorporated an RNN layer in the architecture of both the actor and the critic networks.

The actor network represents the policy $\pi_\theta(a | s')$, where $s' \in S'$, and θ is a the set of the network’s parameters. The input layer is a vector that represents the short term history matrix, of size $|S^s| \times |T^s|$. This vector is passed through an LSTM layer, which is followed by a few linear layers separated by a ReLU activation function. The output of the last layer is a vector of size $|T^s|$. It is passed through a softmax function to give the actual output of the network in a form of a probability distribution over the possible interactions T^s . The parameters of this network are optimized to maximize the objective function L^{CLIP} .

The critic network serves as the estimated state-value function of the current policy π_θ induced by the actor. Its structure is almost identical to the actor network. It differs only in the last layer, which has only one neuron. This single neuron’s value is the approximated value of the input state vector $V(s')$. The estimated advantages are calculated based on the controller state values received from this network, and the reward-to-go values derived from the sampled episodes. The critic parameters are optimized to minimize the *MSE* loss function.

The Training Phase Our goal is to find a set of parameters θ that maximizes the successful number of interactions with the environment. We start by training the actor and the critic networks. At the beginning of every optimization phase of the two networks, we use our current actor network π_θ to sample a constant number of episodes of the same length. At each timestep, the actor network returns a probability distribution over all the possible interactions.

On the one hand, we can exploit the current knowledge of the actor network and always choose the interaction with the highest probability to maximize the number of successful interactions. On the other hand, exploring new interactions with lower probabilities may be necessary to gather information about the environment and discover new and potentially better interactions.

We observed that a decaying level of exploration over time helps in finding optimal policies in our setting. Hence, we chose our next action in the training phase according to the *Boltzmann Exploration* technique. Instead of deriving the distribution over the possible interactions based on the regular softmax function, we used a softmax function with a temperature parameter T .

$$\text{softmax}(t_1, \dots, t_n; T) = \left(\frac{e^{\frac{t_1}{T}}}{\sum_i e^{\frac{t_i}{T}}}, \dots, \dots, \frac{e^{\frac{t_n}{T}}}{\sum_i e^{\frac{t_i}{T}}} \right)$$

We sample the next interaction according to this distribution. The temperature parameter is used to control the degree of exploration, with higher temperatures leading to more exploration and lower temperatures leading to more exploitation. We start the training phase with an initial temperature value of T_0 and gradually decay it in a geometric fashion. We stop decaying it when $T = 1$.

In [7], exploration is performed during training by selecting a random action in generating the current execution of the system; exploitation is done by selecting the action with the highest probability among the probability distribution provided by the network. A constant ϵ represented the probability for exploring the environment. Note that this exploration technique ignores any information the controller network might hold regarding the actions. Even if one action is less probable to select than another action according to the network’s action distribution, the two would be selected with an equal probability as part of an exploration step. On the contrary, our approach uses the actor network to both explore and exploit. By sampling the next action directly from the action distribution, less probable actions according to the actor network would be selected less frequently than the more probable ones.

We assigned two reward values for the offered interactions in the collected episodes. A reward of 1 was given to a successful interaction and a reward of -1 was given otherwise. After optimizing both networks according to their objective functions L^{CLIP} and MSE , a new optimization step begins. Note that we do not directly use the critic network to select an interaction. The critic network is used as a mean to estimate the value of a controller state, as a part of the estimated advantage calculation of every observed state-action pair. The training phase ends upon completing a predefined number of optimization steps.

Another difference between our approach and the previously mentioned approach is the frequency and the timing of the optimization steps. In the previous approach, more local optimization steps are performed. This started with a loss function that was calculated after every interaction with the environment, and optimized the parameters according to its gradient. The loss function definition differed depending on whether the last interaction succeeded or not. However, local optimization steps may not always yield optimal policies. In some cases we need to look beyond the immediate outcome of an interaction to make an informed decision. Consequently, an addition of a *lookahead* parameter allowed them to optimize the network after observing the outcome of a sequence of interactions. Longer lookahead assisted in learning optimal policies for a relatively more complex scenarios. The controller’s network was optimized in the previous work according to this loss function. This optimization approach may be unstable for two reasons. First, a fraction of an execution may be insufficient for optimizing the networks parameters. In some cases, we would know if a certain action paid off only at the end of an execution. Another reason is that information from a single execution may not adequately represent the effectiveness of a given policy; i.e., an excellent policy on average can, by chance, show weak performance.

PPO performs an optimization step only after at least a few full executions are evaluated. This is reflected in the L^{CLIP} objective function, where an empirical expectation over all batch timesteps is taken. The fact that outcome of full executions is considered essentially eliminates the need for lookahead.

The Evaluation Phase After training, we only exploit the actor network to evaluate its performance. At this point, the critic network is not used at all. In our experiments we assumed that the optimal policies are deterministic rather than stochastic. Hence, exploitation in our context would be selecting the action with the highest probability at each timestep. Stochastic policies might be achieved by sampling interactions from the actor network’s distribution.

3.3 An Extension of the Proposed Approach to the Advanced Model

We now generalize our approach to synthesize a control for the more advanced model where components may interact with each other asynchronously, with local actions in addition to their regular way to interact.

Our system consists now of multiple components with extended functionalities that interact with one another; interactions are allowed only between pairs of components. All the components consider each other as a black box. This time, each action may be associated with one of two types:

- *Interactions* - these actions behave as a standard action in the simplified model, and indicate an attempt of two components (agents, processes) to interact.
- *Local actions* - these actions are performed independently with respect to their component. A necessary and sufficient condition for a local action to be triggered, is to be enabled and selected by the respective component.

As mentioned above, the execution is *asynchronous*, with components performing according to their own speeds; each component operates according to its own internal clock, where a fixed time interval represents the duration of every timestep in the execution. Note that different components may have different time intervals. Despite the asynchronous nature of the joint execution, the interaction between components is still established in a handshake form.

At each timestep, a component may offer an interaction to another component, or perform a local action. If an interaction was offered, it would wait the entire timestep for a response. We first examine the case where two components chose to interact with each other. Here, both components would be informed whether or not their interaction was successful. The interaction is considered successful when the same action was proposed by both components. In this case, the two components change their respective states according to the proposed action. If two different actions were offered, the interaction *fails*. Then, both sides uniformly sample at random some enabled local action and move accordingly.

A component that chose to interact but received no feedback until the end of the timestep may either wait for a response in the next timestep or give up. If it gives up, its action would be considered as a *missed* interaction from its perspective, and it would remain at its current state. Obviously, a component that selected a local action does not need a feedback for a its action, since it is always successful regardless of the other components. In this case the component immediately changes its state according to the selected local action.

Note that it is not clear what is the goal of each component in this scenario. Maximizing the individual sum of rewards in this case does not always maximize social welfare. However, when the reward given to a local action is lower than a reward of a successful interaction, then an individual maximization of the accumulated sum of reward should also lead to the optimal result, with respect to the social welfare. We therefore assumed that local actions are always less rewarding than interactions in the more advanced model.

In light of this insight, we modified our approach to adapt our advanced model. We assigned each component’s controller a pair of actor and critic networks, and used the same training routine in parallel with respect to each controller. This setup is often called *decentralized training* [16]. In each one of their optimization steps, the components obtain their collection of episodes by interacting with each other.

When the different controllers are trained in a decentralized fashion, each one of them considers the others as the environment. However, while a standard environment in reinforcement learning has Markovian behavior, our controllers constantly change their strategies throughout the training phase, completely invalidating the Markovian assumption on their strategy.

As a result of the constantly changing behavior of the components in the training phase, we tested different parallel exploration techniques in our experiments. Specifically, we tested the efficiency of different techniques on scenarios with two components. We started by alternately applying the same exploration or exploitation policy for both components. i.e. in each optimization step the two controllers selected their actions according to Boltzmann exploration, or they both exploited their neural network, selecting the most probable action.

We examined an additional exploration technique. According to this technique, in every optimization step, only one of the components performs Boltzmann exploration, while the other always exploits its network. In the next optimization step, the components change their roles with respect to their exploration/exploitation strategies. The idea behind this technique is that the exploring component would be able to learn a more stabilized version of his peer, which in constantly exploiting, thus acting in a more predictable way.

Finally, the most stable results were achieved where both components were exploring according to Boltzmann exploration throughout the entire training phase. It seems that the optimization steps of PPO are small and cautious enough, so both agents can explore simultaneously and still converge to strategies that maximize their individual cumulative rewards with a relatively high probability. We observed than this exploration technique is efficient also for sce-

narios with more than two components. The results are further described in the experiments section.

4 Experiments

We present here the experimental results for our proposed approach. The experiments consist of two parts. In the first part we tested our approach with respect to six existing examples of the simplified model. The second part examines our extended approach in the more advanced setting. All the experiments were conducted on the same machine with an Intel[®] Core[™] i5 CPU at 2.4 GHz and 8GB of RAM, running on a Windows 11 operating system. Our approach was implemented in Python, using the PyTorch library.

4.1 Experiments with the Simplified Model

In [7], six different examples of system and environment pairs to experiment with were studied. The first four are relatively simple. The latter two are more complex, combining the behaviors of two simple examples. The first simple example - *permitted*, was described in Section 2. We present here two additional simple examples.

In example *schedule* in Figure 2, the controller must make sure that the system will never choose an *a*. Otherwise, after interacting on *a*, the environment will progress to e_3 , and no successful interaction with *b* will be available further. A controller with two states that alternates between *b* and *c*, i.e., allows exactly the sequence of interactions $(bc)^*$ is sufficient to guarantee that no failure ever occurs.

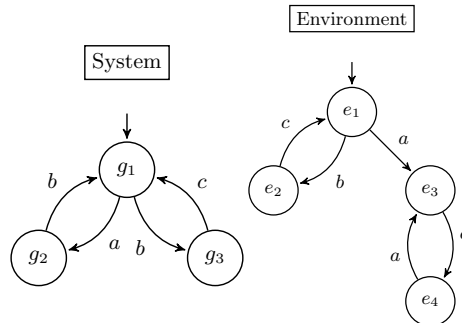


Fig. 2: *schedule*: The control needs to avoid the action *a* as its initial choice.

In the example *cases* in Figure 3, the system is obliged to offer an *a* from its initial state. The environment allows initially only *b* or *c*. Hence, the interaction will fail, the system will stay at the same state and the environment will progress to e_2 or to e_3 , according to its choice, which is not visible to the system. After

the first a , the system does not change state, hence a is again the only action that it offers.

An optimal controller for this example has to consider the execution history to make an informed decision. After the system offers the first a , which is due to fail, it checks whether offering a fails again. If it does, then the next interaction to be offered is c . Otherwise, it selects the action b .

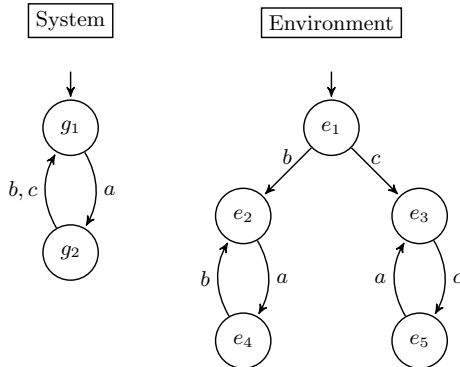


Fig. 3: *cases*: Needs to check if a succeeded.

In their experiments, the hyperparameter values that achieved the optimal results were different for each example. In addition, their training routine for the complex examples was different comparing to their simpler examples. We tested our approach against all six examples. The last complex example (*cycle scc*) assumed to be the most difficult one, since it had the highest average failure rate. Table 1 exhibits the properties of the tested experiments.

Experiment	Environment States	Best Failure Rate (%)
Permitted	3	0
Schedule	4	0
Cases	5	1.5
Choice-scc	25	1.5
Schedule-cycle	4	0
Cycle-scc	25	1.5

Table 1: Experiments list. Failure percentages are per episodes with 200 timesteps.

In our experiments, we used the same training routine for all examples. Moreover, the values of the different hyperparameters in our approach did not vary between different examples. We trained our actor and critic network as presented

in the previous section, with 160 sampled episodes of length of 25 in each optimization phase, for a maximum of 150 optimization phases. In most of the cases, the optimal policy was found in less than 100 optimization steps.

We used *Adam* [8] to optimize the weights of both the actor and the critic networks, with a learning rate of 0.01. Adam is an adaptive learning rate optimization algorithm that has been designed specifically for training deep neural networks. The PPO clipping parameter ϵ was set to 0.2, and the initial exploration temperature was $T_0 = 2$ with a geometric decay rate of 1.008. The discount factor for the calculated reward-to-go was set to $\gamma = 0.99$.

We used the same evaluation metric as [7]. For each example, we trained the model and evaluated it on 100 different episodes of 200 timesteps. We repeated the training and the evaluation process ten times and calculated the average failure rate. Table 2 shows a comparison between the average failure rates. Our approach achieves the same performance as [7] in the first three examples, and outperforms it in the last three examples.

Experiment	Results of [7]	Our PPO based Approach
Permitted	0	0
Schedule	0	0
Cases	0	0
Choice-scc	4.5	1.5
Schedule-cycle	5	0
Cycle-scc	33.5	3

Table 2: Average failure rates (%) - the reinforce-based approach vs. our PPO-based approach.

4.2 Experiments with the Advanced Model

We tested the extension of our approach on examples with multiple components, with both interactions and local actions. The rewards that were given to failed and successful interactions were identical to previous experiments. The reward for local actions was lower than the one given for interactions, and varied between examples. Each example was tested with respect to two different execution profiles. The first profile is the case when the two components perform according to the same speed, while in the second one the execution speed of one components is two times faster than the other throughout the whole execution. Here we did not consider the failure rate, but the average cumulative rewards (CR) throughout the execution.

We started to test our approach on an example called *hold back*. It is described in Figure 4; the action l is a local action in both components, while all the others are interactions. The reward for local actions was set to 0.5, and the reward for

a successful interactions was set to 1. A reward of 0 was given for a missed interaction, while a failed interaction had a reward of -1.

In this example, both components are faced with three consecutive decisions, where they have to choose between two options with different payoffs. The optimal result for them may be achieved only if they both choose the least rewarding course of action at every decision point. In an execution profile where there are different execution speeds, the faster process may have to wait after the three first local actions before successfully triggering the action a .

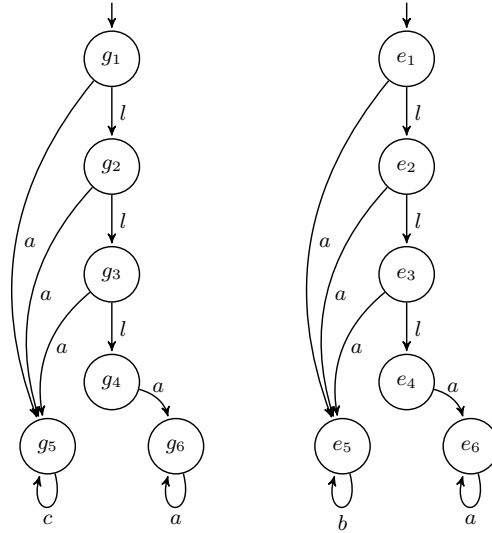


Fig. 4: *hold back*: Three consecutive decisions with different payoffs. The action l is a local action in both components, while all the others are interactions.

The second example we designed is called *abcde coordination*. It is depicted in Figure 5. In this example, the maximal cumulative reward of each process is achieved if both components learn to repeat the following sequence of interactions: $a b c d e$, where l is a local action. It is not a simple task to coordinate this pattern of interactions; both components have some self-loop actions that should be triggered at the right time. Moreover, this task is even more difficult in an execution profile where the two components have different execution speeds. For instance, if the component on the right is faster than the other one, an optimal strategy for it would be utilizing the local actions on its self loops in between interactions, so it can be correlated with the slower component. Here, the rewards for interactions are identical to the hold back example, and the reward for local actions was set to 0.1.

The third example we introduce in Figure 6 is called *wait to succeed*. The actions l, l_1 and l_2 are local actions in this example. All the reward values are identical to the values in *abcde coordination*, except of the local action reward that was set to 0.02. As part of the optimal strategy in this example, both com-

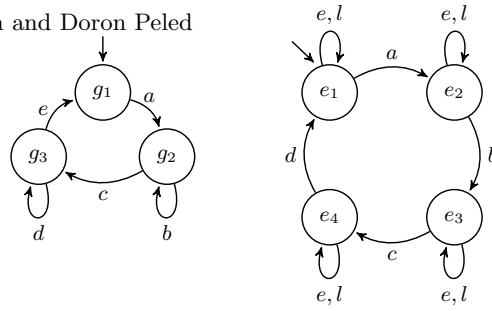


Fig. 5: *abcde coordination*: A certain repeating sequence leads to a non-failure execution. l is a local action.

ponents would aim to end the execution with a sequence of consecutive successful interactions of type d . To achieve that in the equal-speeds execution profile, they would have to coordinate on the sequence $a b c$, and then trigger two local actions before initiate an interaction of type d . Both the preliminary sequence and the two local actions waiting are necessary towards having a successful sequence of interactions of type d . Now, the optimal strategy of the left hand side component changes assuming it is faster than the other component. It has to utilize the local actions of from states g_2, g_3 and g_4 to adapt the slower component.

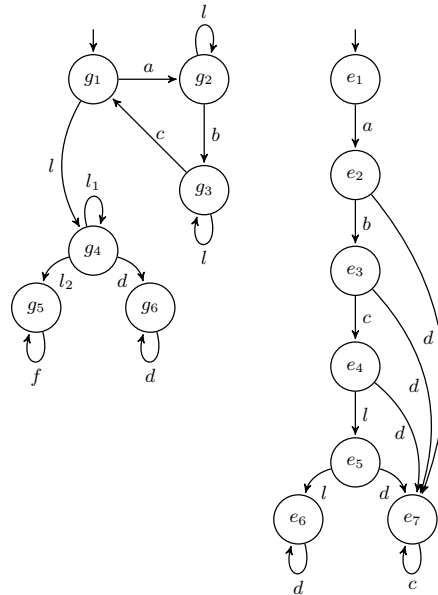


Fig. 6: *wait to succeed*: The two components will have to coordinate with respect to a specific sequence, then trigger a few local actions before they may continue with a long sequence of consecutive successful interactions of type d . The actions l, l_1 and l_2 are local actions.

The fourth example, which is called *triple coordination*, consists of three asynchronous components. It is described in Figure 7. The actions l, l_1, l_2 are local actions, while all the others are interactions. The reward for local actions was set to 0.1, and the reward for a successful interaction was set to 1. A reward of 0 was given for a missed interaction, while a failed interaction had a reward of -1. In this example, component (a) is able to interact with both components (b) and (c). However, components (b) and (c) cannot interact with each other. In order to achieve an optimal performance, component (a) would aim to end its execution with a sequence of alternating interactions from s_1 , where f is offered to component (b), and e is offered to component (c). This execution suffix would have been possible only if component (a) and (b) had coordinated on the sequence $a b c$ at the beginning of the execution. We denote the first execution profile by p_1 . In this profile, the components have equal speeds. In the second execution profile, denoted by p_2 , components (a) and (c) have equal speeds, while component (b) is two times slower.

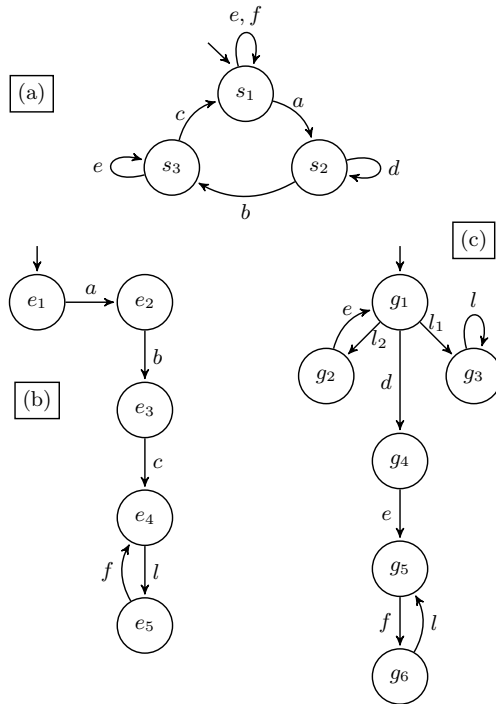


Fig. 7: *triple coordination*: The actions l, l_1, l_2 are local actions, while all the others are interactions. Component (a) is able to interact with both (b) and (c). However, components (b) and (c) cannot interact with each other.

Most of the hyperparameter values were identical to previous experiments. We slightly changed the initial exploration temperature to 2.5, and the geometric

decay rate to 1.01. As mentioned before, both components were exploring according to Boltzmann exploration throughout the entire training phase. Again, we trained the controllers and evaluated them on 100 different episodes of 200 timesteps. We repeated the training and the evaluation process ten times.

Experiment	Optimal CR		Our Avg. CR		Our Med. CR	
	$Cp.a$	$Cp.b$	$Cp.a$	$Cp.b$	$Cp.a$	$Cp.b$
hold-back- p_1	198.80	198.80	155.20	155.20	198.80	198.80
hold-back- p_2	98.80	98.80	98.80	98.80	98.80	98.80
<i>abcde</i> - p_1	200	200	163.20	163.40	200	200
<i>abcde</i> - p_2	100	101	67	67.74	95.60	96.01
w.t succeed- p_1	198.04	198.04	158.80	158.02	197.02	197.02
w.t succeed- p_2	98.14	98.04	96.85	96.83	97.02	97.02

Table 3: Average and median cumulative rewards - a model with two asynchronous components with local actions.

Table 3 exhibits the experimental results of the first three examples. p_1 and p_2 are referred to as the first and the second execution profiles. The table shows that the average cumulative rewards in most of the experiments are lower than the optimal ones, while the median cumulative rewards are close or equal to the optimal values. The reason behind this is the fact that in some repetitions, the interfaces could not learn their optimal strategies, and had a sub-optimal performance.

In the *hold back* example, the optimal strategy was learned in most of the repetitions. In the second execution profile, the faster component learned to trigger its three local actions and then to wait for a first successful interaction of type a .

In *abcde coordination*, the optimal strategy was achieved in most of the repetitions for the first execution profile. In the second execution profile, the right component was selected as the faster component. In this profile, the majority of executions ended with a relatively high cumulative reward. In some cases, the faster component learned to intermittently select a local action in order to adapt the slower component. In other cases, it preferred to wait for a successful interaction rather than intersperse a local action in between.

In the example *wait to succeed*, the two components managed to learn the behavior that leads to the optimal results in most of the cases. In the second execution profile, we selected the left component to be the faster one. In this profile, a common behaviour for the left component was to select local actions before trying interactions from states g_2 and g_3 . However, when being in state g_4 , the typical behaviour was to wait for a successful interaction of type d .

The performance of our extended approach on the triple coordination example is described in Table 4. It shows a comparison between the optimal strategy’s cumulative rewards and the average cumulative rewards according to our

extended approach. It is observed that in the first execution profile, the three components manage to learn the behavior that leads to the optimal result. In the second execution profile, component (a) learns a strategy that is slightly less beneficial than its optimal strategy. Despite having different speeds, component (a) learns how to alternately interact with components (b) and (c) in the suffix of the execution.

Experiment	Optimal CR			Our Avg. CR		
	<i>Cp.a</i>	<i>Cp.b</i>	<i>Cp.c</i>	<i>Cp.a</i>	<i>Cp.b</i>	<i>Cp.c</i>
triple coord.- p_1	200	191.9	189.9	200	191.9	189.9
triple coord.- p_2	152	55.8	109.8	151	55.8	109.8

Table 4: Average cumulative rewards - a model with three asynchronous components with local actions.

5 Conclusion

We presented a deep reinforcement learning approach that constructs control for concurrent components of a system that are capable of interacting with one another, with a limited knowledge of one another’s structure and execution history. We started by designing an approach that improved on the REINFORCE-based method used in [7] for the simplified model.

Our approach is an Actor-Critic implementation of the PPO algorithm with some modifications. We have distinguished between two types of execution histories: short and long term histories, and handled them in different ways. The short term history was encoded as part of the control’s state space representation, while the long term history was captured by the recurrent layers we have integrated in both the actor and the critic networks. In addition, we found an exploration strategy that suited all the tested examples in our experiments. In the tested experiments, not only did our approach outperform the method presented in [7], but it was capable of using a single set of hyperparameter values and training routine for all tested examples.

We have suggested an architecture for optimizing the integration of systems that consist of multiple processes. According to this learning-open architecture, each component is equipped with a local neural network that can be trained to control its behavior. As the different components can be constructed separately, the training of the system does not involve the sharing of information between the deep learning components; each process (and its controller) is aware only of its own states, and the outcome of the interactions. Hence, deep reinforcement learning methods such as DQN [11], which focus on estimating the utility value of each state in the state space are less suitable for our task. We conclude that

policy gradient methods such as PPO are more adequate in case of a state space with limited observability.

Our approach appears to be stable with optimization steps that are relatively small and rely on full sampled episodes. As opposed to the previously mentioned approach that relies on a single episode each time, in our approach each optimization step may be more significant. Moreover, the nature of PPO inherently eliminates the use of a lookahead. We do not need to consider a limited horizon and optimize the parameters after every timestep, it can be done once after generating a batch of full episodes. On the other hand, in most of the cases, our approach would need to sample relatively more executions to construct an effective strategy. Apparently, in this case, the effectiveness and the robustness of our approach in different scenarios come at a cost.

We realized that our approach requires only a few minor modifications to adapt to the more advanced setting, even though the Markovian assumption no longer holds for each component. In the examples we tested, the components manage to converge towards an optimal outcome with high probability, despite the scarce signals that are transmitted between the different components. Future directions of this research include applying this approach to more complex components with more states and extended functionality. In addition, our approach should also be evaluated in settings where maximizing the individual objective may not lead to the optimal global outcome, e.g., when two or more components have conflicting objectives.

References

1. Brown, T.B., et al.: Language models are few-shot learners. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (2020)
2. Busoni, L., Babuska, R., Schutter, B.D.: A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part C* **38**(2), 156–172 (2008)
3. Cassandras, C.G., Lafortune, S.: *Introduction to Discrete Event Systems*, Second Edition. Springer (2008)
4. Gößler, G., Sifakis, J.: Composition for component-based modeling. *Sci. Comput. Program.* **55**(1-3), 161–183 (2005)
5. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: *International conference on machine learning*. pp. 1861–1870. PMLR (2018)
6. Hoare, C.A.R.: Communicating sequential processes. *Commun. ACM* **21**(8), 666–677 (1978)
7. Iosti, S., Peled, D., Aharon, K., Bensalem, S., Goldberg, Y.: Synthesizing control for a system with black box environment, based on deep learning. In: *International Symposium on Leveraging Applications of Formal Methods*. pp. 457–472. Springer (2020)
8. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014)
9. Kravaris, T., Vouros, G.A.: Deep multiagent reinforcement learning methods addressing the scalability challenge. *Appl. Intell.* (2023)

10. Lehmann, D., Rabin, M.O.: On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 133–138 (1981)
11. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
12. Paszke, A., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
13. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
14. Sutton, R.S., Barto, A.G.: Reinforcement learning - an introduction, 2nd Edition. Adaptive computation and machine learning, MIT Press (2018)
15. Tan, M.: Multi-agent reinforcement learning: Independent versus cooperative agents. In: Utgoff, P.E. (ed.) Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993. pp. 330–337. Morgan Kaufmann (1993)
16. Zhang, K., Yang, Z., Liu, H., Zhang, T., Basar, T.: Fully decentralized multi-agent reinforcement learning with networked agents. In: International Conference on Machine Learning. pp. 5872–5881. PMLR (2018)