Scaling Directed Controller Synthesis via Reinforcement Learning

Tomás Delgado¹, Víctor Braberman¹, Sebastián Uchitel¹,²

¹Universidad de Buenos Aires, Argentina
²Imperial College of London, United Kingdom
tdelgado@dc.uba.ar, vbraber@dc.uba.ar, suchitel@dc.uba.ar

Abstract
Directed Controller Synthesis technique finds solutions for the non-blocking property in discrete event systems by exploring a reduced portion of the exponentially big state space, using best-first search. Aiming to minimize the explored states, it is currently guided by a domain-independent handcrafted heuristic, with which it reaches state-of-the-art performance.

In this work, we propose a new method for obtaining heuristics based on Reinforcement Learning. The synthesis algorithm is framed as an RL task with an unbounded action space and a modified version of DQN is used. With a simple and general set of features, we show that it is possible to learn heuristics on small versions of a problem in a way that generalizes to the larger instances. Our agents learn from scratch and outperform the existing heuristic overall, in instances unseen during training.

1 Introduction
Discrete event control theory aims at automatically constructing a controller for an environment modelled as a Discrete Event System (DES) (Cassandras and Lafortune, 2006), also called a plant. A controller dynamically disables controllable events while monitoring uncontrollable events in order to restrict the plant’s behaviour to satisfy given constraints. Automated synthesis of controllers given a formal specification of a system has been extensively studied in the software engineering community for some time.

Critical dynamic systems such as communication networks, automated manufacturing, air traffic control, etc., have only become more common with the evolution of computing technologies, and errors in these systems can have catastrophic consequences. This makes automated and correct synthesis methods very appealing. However, synthesis methods are usually limited by a state explosion problem, and scaling them to larger problems represents a research challenge that is receiving increased interest.

On the other hand, Reinforcement Learning (RL) has exhibited remarkable empirical success in recent years as a general approach for sequential decision-making. With the seminal work in which DQN was proposed for the Atari games (Mnih et al., 2013) and the exceptional results for the games of Go, Chess and Shogi (Silver et al., 2017), the field has shown great capabilities for learning from scratch in intractable state-spaces. While RL can find very good policies for a wide range of problems, it is not directly adequate for the problems that controller synthesis aims to solve, where guarantees of correctness are desired. The approach that we present is one way of joining the best of both worlds, using RL as a heuristic to accelerate a correct and complete synthesis algorithm, guiding the exploration of the exponentially big state space.

In particular, we focus on Directed Controller Synthesis (DCS) for controllers that are safe and nonblocking (Huang and Kumar, 2008). Directors are controllers that select at most one controllable event for each state, as opposed to the common but more expensive notion of maximally permissive supervisors (W. M. Wonham and P. J. Ramadge, 1987). Controllers are safe if they guarantee that no deadlock (or error) state can be reached, and they are nonblocking if they guarantee that a marked target state can always eventually be reached.

The plant (DES) can be modelled modularly as the parallel composition of finite state automata that describe individual interacting components (Wonham and Ramadge, 1988). However, the underlying semantics is the monolithic product space that can grow exponentially with the number of intervening components and with the size of the components. Instead of building the full DES and then solving the control problem, the on-the-fly DCS algorithm (Ciolek et al., accepted 2022) attempts to avoid or delay the state-explosion by exploring the composed plant incrementally, checking for solutions after each new transition is added. This process, if joined with a good search heuristic, can potentially solve DCS problems by building only the transitions needed in a winning control strategy (or those needed to show that it is impossible).

The Ready Abstraction (RA), proposed with the on-the-fly algorithm, is currently the best heuristic and for some domains it has allowed solving many instances that were not solvable by building the full plant (with reasonable amounts of memory). RA uses a dependency graph to estimate the distance to a marked state for the available transitions, exploiting the modular description of the problem, but it fails to consider other sources of information that could be crucial for finding a winning strategy, such as controllability, already explored states, etc. Adding these considerations manually is a complex engineering task with many unclear trade-offs.
which could even depend on the problem being solved. In this work we study the possibility of replacing the RA heuristic with a Reinforcement Learning approach that learns from scratch. The on-the-fly algorithm itself is casted as a Markov Decision Process in which reward is obtained by minimizing the number of explored transitions of the DES. This task can be challenging because (1) it has a sparse reward, (2) while state-of-the-art Deep RL algorithms usually assume a fixed action set, the action set for this task changes in each state and can grow unboundedly, and (3) the state has a complex graph structure.

Furthermore, RL has traditionally focused on tasks where agents are evaluated in the same environment as they are trained (Kirk et al., 2022). This is not possible in our application because we are ultimately interested in solving large instances that cannot currently be solved in reasonable amounts of time, and training requires solving an instance many times. Thus, the main concern of this work lies in finding policies that generalize to unseen instances.

Specifically, we perform zero-shot policy transfer, since we use the policy learned in one environment in a different (but similar) environment, without re-training. We propose training in a small instance of a problem for a relatively short time with a small neural network and a general set of features. Then, a selection step is used to find the heuristics that generalize best and avoid overfitted policies. Our agents outperform the existing non-learning heuristic overall, pushing the boundaries of instances solved in a subset of the benchmark problems.

2 Related Work

A similar approach has been recently considered for classical planning (Gehring et al., 2021). It also trains an RL agent to guide the search and studies generalization of the learned heuristics to larger instances. The main difference (apart from those between planning and supervisory control) is that their agents solve a task in which the states are the same as the states of the problem being solved whereas our agents solve a task in which the states are explored subgraphs of the state space. Their agents select actions to reach a goal efficiently (backtracking when necessary) while ours decide which actions to explore, including those controlled by the environment, until all necessary transitions to guarantee a liveness goal have been explored (or to prove that there is none). Another difference is that they learn residuals on existing heuristics using reward shaping to accelerate the learning process in what otherwise would be a sparse-reward environment. We learn from scratch in a sparse-reward environment.

Thus, the main concern of this work lies in finding policies that generalize to unseen instances.

Finally, synthesis problem in general (i.e., automatically constructing control rules from a specification), has been addressed by different communities such as Discrete Event Control (Ramadge and Wonham, 1987), Reactive Synthesis (Pnueli and Rosner, 1989) and Automated Planning (Nau, Ghallab, and Traverso, 2004). They consider distinct perspectives on representational and computational aspects. Interestingly, the three disciplines share the important characteristic that input specifications are given via compact descriptions (i.e., the problems’ semantics are based on sorts of transitions systems, which are often exponential with respect to the size of these compact descriptions). This has led to studies that relate the fields (Ehlers et al., 2016; Schmuck, Moor, and Majumdar, 2019; Sardinha and D’Ippolito, 2015) and for test case prioritization (Bagherzadeh, Kahani, and Briand, 2020; Spieker et al., 2017).

Finally, the synthesis problem in general (i.e., automatically constructing control rules from a specification), has been addressed by different communities such as Discrete Event Control (Ramadge and Wonham, 1987), Reactive Synthesis (Pnueli and Rosner, 1989) and Automated Planning (Nau, Ghallab, and Traverso, 2004). They consider distinct perspectives on representational and computational aspects. Interestingly, the three disciplines share the important characteristic that input specifications are given via compact descriptions (i.e., the problems’ semantics are based on sorts of transitions systems, which are often exponential with respect to the size of these compact descriptions). This has led to studies that relate the fields (Ehlers et al., 2016; Schmuck, Moor, and Majumdar, 2019; Sardinha and D’Ippolito, 2015) and for test case prioritization (Bagherzadeh, Kahani, and Briand, 2020; Spieker et al., 2017).

3 Background

3.1 The Modular Directed Control problem

A Discrete Event System to be controlled, called a plant, is modelled with a deterministic automaton with marked states and events that can be either controllable or uncontrollable. Marked states are used to indicate the termination of a task that has to be completed repeatedly (Huang and Kumar, 2008).

More formally, a tuple $E = (S_E, A_E, \to_E, \bar{s}, M_E)$ is a DES if $S_E$ is a finite set of states; $A_E$ is a finite event set, disjoint union of $A^c_E$ and $A^u_E$, the sets of controllable and uncontrollable events; $\to_E: S_E \times A_E \rightarrow S_E$ is a partial function; $\bar{s} \in S_E$ is the initial state; and $M_E \subseteq S_E$ is a set of marked states.

This automaton defines a language $L(E) \subseteq A^*_E$, where $*$ denotes the Kleen closure, in the usual manner. A word $w \in A^*_E$ belongs to the language if it follows $\to_E$ with a sequence of states $\bar{s} = s_0 \ldots s_t$. In this case we note $\bar{s} \xrightarrow{w} s_t$. 

\[ A^*_E = \{ w \mid w \in A^*_E \} \]
A function that based on the observed behaviour of a plant decides which controllable events are allowed is referred to as a control function. Given a DES, a controller is a function σ: A_E → Π(A_E). A word w ∈ L(E) belongs to L_σ(E), the language generated by σ, if each event t_i is either uncontrollable or enabled by σ(l_0, ..., l_{i-1}).

Given a plant, we wish to find a controller that ensures that a marked state can be reached from any reachable state (even from marked states). In particular, the controller should guarantee that uncontrollable actions cannot lead the system to deadlock states. The non-blocking property captures this idea. Formally, a controller σ for a given DES is Non-blocking if for any trace w ∈ L_σ(E), there is a non-empty word w' ∈ A_E such that the concatenation w+w' ∈ L_σ(E) and s ⊢ w+w' s_m for some s_m ∈ M_E. Additionally, a controller is a director if |σ(s)| ≤ 1 for all s ∈ S_E.

Note that a non-blocking controller must also be safe in the sense that it cannot allow a deadlock state to be reachable (i.e. a state with no outgoing transitions). Error or illegal states can be modelled as deadlock states (Magee and Kramer, 2014).

Modular modelling of DES control problems (Ramadge and Wonham, 1989) supports describing the plant by means of multiple deterministic automata and their synchronous or parallel composition.

The parallel composition (||) of two DES T and Q yields a DES T || Q = (S_T × S_Q, A_T ∪ A_Q, →_T || Q, {t, q}, M_T × M_Q), where A_T||Q = A_T ∪ A_Q and →_T||Q is the smallest relation that satisfies the following rules:

(i) if t →_T t’ and l ∈ A_T\A_Q then (t, q) →_T||Q (t’, q),
(ii) if q →_Q q’ and l ∈ A_T ∩ A_Q then (t, q) →_T||Q (t’, q’),
(iii) if t →_T t’, q →_Q q’, and l ∈ A_T ∩ A_Q then (t, q) →_T||Q (t’, q’).

Finally, a Modular Directed Control Problem, or simply control problem in this paper, is given by a set of deterministic automata E = (E_1, ..., E_n). A solution to this problem is a non-blocking director for E_1 || ... || E_n. We abuse notation and use E = (S_E, A_E, →_E, e, M_E) to refer to the composition E_1 || ... || E_n.

3.2 On-the-fly algorithm for the Modular Directed Control Problem

The Modular Directed Control Problem can be solved by fully building the composed plant and running a monolithic algorithm such as the presented by Huang and Kumar (2008). While this quickly becomes intractable, there are problems for which the state explosion can be delayed significantly by exploring a small subset of the plant that is enough to determine a winning strategy (or conclude that there is none). The OTF-DCS algorithm (Ciolek et al., accepted 2022) is briefly summarized in Algorithm 1. It incrementally explores the composed plant adding one transition at a time from the exploration frontier to a partial exploration structure.

Given E = (S_E, A_E, →_E, e, M_E), a control problem, and h = (a_i)_{i=0}^t ⊆→_E, a sequence of transitions, we define the exploration frontier of E after expanding sequence h as F(E, h), the set of transitions (s, ℓ, s’) ∈ (∆_E \ h)

such that s = s or (s”, ℓ, s) ∈ h for some s”. An exploration sequence for E is \{a_i\}_{i=0}^t ⊂→_E such that a_i ∈ F(E, {a_j}_{j=0}^{i-1}) for 0 ≤ i ≤ t.

Algorithm 1 On-the-fly exploration procedure.

Given E a control problem and H a heuristic
h ← Empty list.
ES ← (\{s\}, A_E, {}, s, M_E \ s) WinningStates ← φ
LosingStates ← φ

while \* ∈ WinningStates \cup LosingStates do
a ← H(E, h)
expandAndPropagate(a, ES, WinningStates, LosingStates)
Append a to h
if \* ∈ WinningStates then
return buildController(h, WinningStates)
else
return UNREALIZABLE

After each addition, expandAndPropagate ensures that all expanded states are correctly and completely classified into sets of states that are losing, winning, or neither. We say that a state s ∈ E is winning (resp. losing) in a plant E if there is a (resp. there is no) solution for E_s, where E_s is the result of changing the initial state of E to s. Essentially, a state will be winning if it is part of a loop that has a marked state and that has no uncontrollable events that go to states outside the loop, or if it can controllably reach a winning state. A state will be losing if it has no path to a marked state, if it can be forced by uncontrollable events towards a losing state, or if all its events are controllable but they lead to losing states. For a given (uncompleted) exploration sequence a state is defined to be winning (resp. losing) if it is winning (resp. losing) when assuming that every transition in the exploration frontier goes to a losing (resp. winning) state.

The completeness of the propagation procedure guarantees that a verdict for the initial state will be found after the last transition of the composition is expanded. Hence, a heuristic will try to minimize the number of transitions explored, but even with very poor decisions, the correctness of the synthesis algorithm is not threatened.

A heuristic is a function H(E, h) that observes a control problem E and an exploration sequence h for E and outputs a transition (s, ℓ, s’) ∈ F(E, h). Given a control problem E, the associated exploration optimization problem consists of finding a heuristic H that minimizes the number of transitions expanded by OTF-DCS at problem E, if transitions are selected using H.

3.3 Q-Learning with function approximation

Reinforcement Learning considers an agent that interacts iteratively with its environment, learning to maximize a reward function. An episodic RL task can be formalized as a Markov Decision Process (MDP) M = (S, A, P, R, S_0) where S is a set of states; A is a set of actions; P : S × A × S → [0, 1] encodes the probability Pr[s’|a, s] of observing state s’ after selecting action a in state s; r : S × A × S → R is a reward function; and S_0 is the initial state. The set of available
actions in state $s$ is denoted by $A(s)$.

An episode is a sequence of interactions that starts with the initial state and ends with a terminal state. The behaviour of an agent is defined by a policy $\pi: S \times A \mapsto [0, 1]$ that represents the probability of taking action $a$ after observing state $s$ (0 if $a \notin A(s)$). It is possible then to define random variables $T, S_t, A_t$ and $R_t$, for the number of steps in the episode, state, action and reward at step $t$, respectively. The goal is to find a policy $\pi$ that maximises the expected accumulated reward $E_{\pi} \left[ \sum_{t=0}^{T} R_t \right]$.

Q-Learning (Watkins and Dayan, 1992) is a widely used learning algorithm that approximates an optimal action-value function $Q^* : S \times A \mapsto \mathbb{R}$, which satisfies

$$Q^*(s, a) = \max_\pi \mathbb{E}_\pi \left[ \sum_{t=0}^{T} R_t \mid S_t = s, A_t = a \right].$$

That is, the expected accumulated reward that is obtained after taking action $a$ in state $s$ and then following an optimal policy. Any action-value function directly induces a greedy policy that always chooses the action that maximizes $Q$ for the given state, and the policy induced by $Q^*$ is an optimal policy. To learn this function, one-step Q-learning in its tabular form uses the following temporal difference update at timestep $t$:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \delta_t,$$

$$\delta_t = R_{t+1} + \max_{a \in A_{S_{t+1}}} Q(S_{t+1}, a) - Q(S_t, A_t)$$

where $\alpha$ is a step-size parameter. This value-iteration method guarantees convergence to $Q^*$ with probability one in MDPs, under the assumption that every state-action pair is visited infinitely many times and $\alpha$ is reduced appropriately. However, updating one state-action pair at a time is impractical with large state-action spaces, and generalization across states and actions is needed. This method is commonly replaced by a function approximator $\hat{Q}(s, a, w)$, where $w$ is a parameter vector. If $\hat{Q}$ is a linear function or a neural network, gradient-descent can be used to minimize the squared error, obtaining the following update rule (with the target $\delta_t$ as before):

$$w_{t+1} = w_t + \alpha \delta_t \nabla_w \hat{Q}(S_t, A_t, w_t).$$

(1)

Q-Learning usually follows an $\varepsilon$-greedy policy during training, making it an off-policy algorithm. It takes the action selected by $\hat{Q}$ with probability $1 - \varepsilon$ and a random available action otherwise.

Using temporal-difference updates, function approximation and off-policy learning as this approach does has been called a deadly triad, since it is known that it can generate unstable learning and divergent behaviours (e.g. see Chapter 11 of Sutton and Barto 2018). Two techniques that have been proposed to restore stability are Experience Replay and Fixed Q-Targets (Mnih et al., 2013; Lin, 1992). The first one consists in updating with batches of observations sampled randomly from a buffer that contains the last $B$ iterations. The second one proposes to use a target neural network for the right-hand side of equation 1, updating it as a copy of the current network with a fixed frequency.

4 Methods

4.1 Formulating On-the-fly Directed Controller Synthesis as an MDP

In this section we define an MDP whose solution can be used for the Exploration Optimization Problem. Given the similarity between MDPs and DES, it is tempting to define an MDP in a straightforward manner, where states and actions in the automaton correspond to states and actions in the MDP, and a reward in the MDP is given when a marked state is visited in the automaton. Then, an RL agent playing in that MDP could learn to reach marked states rapidly. However, this is not what we want because (1) events in the plant can be uncontrollable and we have to explore them to prove that we can reach our goal even if the environment takes them and because (2) reaching a marked state is not sufficient, we need to know how to continue exploring to find a controllable loop that contains that state. In our MDP, a state is defined as the state of the exploration process, and an action represents the expansion of a transition. A terminal state is an exploration state in which the initial problem state is marked as winning or losing.

Let $E = (E_1, \ldots, E_n)$, a control problem. We define the associated RL task as a deterministic MDP $(S, A, P, r, S_0)$ where:

- $S = \{h, E : h$ is an exploration sequence for $E\}$. Since $E$ is constant, when it is clear from the context we use $s \in S$ as an exploration sequence.
- $A = S_E \times A_E$.
- $P(s, a, s') = 1$ if $a \in F(E, s)$ and $s' = sa$, and 0 otherwise.
- $r(s, a, s') = -1 \forall s, a, s'$.
- $(s, E) \in S$ is a terminal state if the initial state is marked as winning or losing after expanding sequence $s$ in $E$.
- $S_0 = (\emptyset, E)$.

Three difficulties arise when trying to learn a policy for this task:

(P1) The defined task serves as an underlying MDP, where the state includes all the information that could be available for an agent, but it is clearly impractical. It would be very difficult for an agent to use the full history directly. Another maybe more natural way to define the states, which also satisfies the Markov property, is the explored subgraph of the plant, possibly including winning and losing states information. Unfortunately, this is also quite complex for a traditional neural network to process. Instead, our agents observe a feature vector that describes the current exploration state and can be computed as a function of the exploration sequence. For this more practical state signal the Markov property does not hold anymore, and the task formally becomes a Partially observable MDP (POMDP).

(P2) The set of available actions is different for each state, both in size (the size of the exploration frontier changes) and in semantics (the 4th action might refer to a completely different transition in two consecutive steps). Additionally,
the number of actions could be very large, since it is only bounded by the number of transitions in the plant. In fact, it is unbounded for the end-to-end problem discussed in section 4.3, where the same agent is expected to play in arbitrarily larger versions of the training task. In particular, this makes most state-of-the-art RL algorithms impossible to apply directly, since they usually rely on a fixed set of actions.

(P3) Since rewards are always \(-1\), the task suffers from the well-known sparse-reward problem. New reward information is only received in the last step of each episode, where the next state has value 0 instead of a bootstrapped value from the neural network. This could make learning inefficient, since it is hard to learn with a long delay between good decisions and their rewards.

4.2 A learning algorithm for the Directed Controller Synthesis RL Task

In this section we describe how neural network-based Q-Learning can be used to solve our RL task for a given control problem. The idea is very similar to that of DQN (Mnih et al., 2013), but it differs in that we evaluate each vector \(\phi\) the new state is terminal the synthesis process is restarted.

The target function is reset with a new copy of \(Q\) saved instead of \(r\). Experience is added to the replay buffer environment propagates the verdicts of winning and losing is evaluated using \(Q\), and an \(\epsilon\)-greedy action is selected. The algorithm 4.2. First, a neural network \(Q\) is initialized with input dimension \(d_E\) and a target network \(E\) is initialized using observations from a policy that chooses a random transition at every step. Then, the agent synthesizes the same problem repeatedly until the time steps run out. At each step \(t\), the feature vector \(\phi(S_t, a)\) of each transition \(a\) in the exploration frontier is evaluated using \(Q\), and an \(\epsilon\)-greedy action is selected. The environment propagates the verdicts of winning and losing states in the explored plant accordingly and the new experience is added to the replay buffer \(B\), removing the oldest experience if necessary. At an implementation level, a vector of feature vectors, one for each transition \(a \in A(S_{t+1})\), is saved instead of \(S_{t+1}\).

After every step a minibatch update is performed on \(Q\) from a random sample of the experience buffer. The target value \(\delta_j\) for experience \((\phi(S_j, a_j), S_{j+1})\) is, if \(S_{j+1}\) is not terminal, the value of the best feature vector in \(S_{j+1}\), according to \(Q'\), minus one (the reward). Once every \(C\) time steps the target function is reset with a new copy of \(Q\). Finally, if the new state is terminal the synthesis process is restarted.

Asymptotically, the evaluation of the neural network does not induce an overhead since the complexity of each iteration of OFT-DCS (expanding a transition and propagating the verdicts) is bounded by \(O(|S_{ES}|^2 \times |A_E|)\) and the number of transitions in the frontier is bounded by \(O(|S_{ES}|^2)\). In practice, the worst-case bound for the propagation procedure could be reached rarely, and the evaluation of a large neural network could add a significant overhead.

4.3 End-to-end approach: Generalizing to larger instances

Training with the instances that we are interested in solving is impractical, not only because learning there would be hard since they are very large MDPs with sparse rewards, but because by definition we want to solve instances that cannot be solved in a reasonable amount of time, and training requires solving an instance repeatedly.

First, we define the problems that we aim to solve. A parametric control problem with \(d\) dimensions is a problem II with instances \(E_p\) for \(p \in \mathbb{N}^d\), where each \(E_p\) is a control problem. In particular, we focus on parametric control problems for which the size of the composed plant grows exponentially with each dimension. Additionally, we expect different instances of the same problem to have some degree of similarity in their structures. This homogeneity hypothesis in our case is supported by the fact that all instances \(E_p\) are defined with the same specification, that uses the dimension \(p\) as an input parameter, using the FSP language (Magee and Kramer, 2014).

Our approach for solving the largest possible instances of a given problem II consists of three steps:

1. Training in an instance \(E_p\) (as described in section 4.2), saving \(N\) sets of weights sampled uniformly.
2. Testing the policies obtained during step 1 on each instance \(E_p\) with a small time-out, only testing instances for which the immediate smaller instances have been solved in time.
3. The policy that generalized best (i.e. solved the most instances, breaking ties with total expanded transitions) is

### Algorithm 2 Q-Learning with function approximation for the Modular Directed Control RL task.

Given a control problem \(E\) and \(T\) a budget of time steps.

1. \(Env \leftarrow \text{off-dcs solver environment for } E\).
2. Initialize neural network \(Q\) with random weights and input dimension \(d_E\).
3. Initialize \(Q'\) as a copy of \(Q\).
4. Initialize buffer \(B\) with observations from a random policy.
5. \(S_0 \leftarrow \text{reset}(Env)\).
6. for \(t = 0\) to \(T\) do
   - \(a_t \leftarrow \begin{cases} \text{a random action} & \text{with probability } \varepsilon \\ \arg\max_{a \in A(S_t)} Q(\phi(S_t, a)) & \text{otherwise} \end{cases}\)
   - \(S_{t+1} \leftarrow \text{Expand and propagate } a_t\) at \(Env\).
   - Add \((\phi(S_t, a_t), S_{t+1})\) to \(B\).
   - Sample transitions \((\phi(S_j, a_j), S_{j+1})\) randomly from \(B\).
   - \(\delta_j \leftarrow -1 + \begin{cases} 0 & \text{if } S_{j+1} \text{ is terminal} \\ \max_{a \in A(S_{j+1})} Q'(\phi(S_{j+1}, a)) & \text{otherwise} \end{cases}\)
   - Perform a gradient descent step on \(Q\) with minibatch \((\phi(S_j, a_j), \delta_j)\).
   - \(Q' \leftarrow Q\) if a fixed number of steps has passed.
   - \(S_{t+1} \leftarrow \text{reset}(Env)\) if \(S_{t+1}\) is terminal.
tested with a larger time-out, obtaining the end-to-end measure of performance.

In this paper we train with only one instance of a fixed size for all problems, but a round robin or incremental training from multiple instances could also be possible. Adding such diversity in the training set has been shown to reduce overfitting (Zhang et al., 2018), but it is not straightforward to do it in our case because the scale of the reward changes significantly in different instances and a $Q$ function that estimates both simultaneously could be a bad policy for all policies individually.

Although our hypothesis of homogeneity suggests that good performances in the training instance correlate in some hard-to-specify way to good performances in larger instances, it is clearly possible for a given set of weights to be an exception to this idea. A policy could be overfitted to the training instance in two ways. First, it could only make good decisions in its deterministic trajectory for that instance, performing poorly if forced to play from any other state of that instance, as has been shown to be possible in the arcade learning environment Machado et al. (2017). Second, and maybe more concerning in our case, an agent could learn a robust strategy that relies on specific characteristics of the training instance and does not generalize well to the larger versions. The testing step (step 2) is important to account for the potential diversity of generalization capabilities of the trained agents.

Another factor that might have a positive impact on generalization is stopping training relatively early. This could be useful following the common idea from supervised learning of stopping training when the performance in the testing set starts decreasing. Nevertheless, performance in our case is quite noisy and we have not found strong evidence for that phenomenon being clearly replicated. Another similar but slightly simpler reason to stop training early is that policies might be more diverse during the first stages of training, before convergence is achieved, making the probability of finding a good general strategy there higher.

### 4.4 Definition of a feature vector

The definition of a set of features that compose the state-action signal and describe the transitions in the frontier and the general state of the exploration is a key component of this approach. The feature function $\phi : S \times A \mapsto \mathbb{R}^{d_{E_p}}$ should be:

1. Informative enough to allow good policies.

2. Independent of instance size: $d_{E_p}$ (the number of features) must be constant for all $E_p$ in a problem $P$ (but not necessarily for different problems). Also, the distribution of each feature should change as little as possible for different instances $E_p$, at least we want the range to be maintained.

3. Not too many and not too computationally expensive, since at every step all features need to be computed and all transitions need to be evaluated.

4. Automatically extracted from $E$, features should not be defined manually for each domain.

Although real-valued features are possible, they generally rely on the neural network generalizing to unseen values during testing, making generalization harder. The feature vector used in this paper is solely composed of boolean features. For a transition $(s, \ell, s')$ and a parametric control problem $E_p = (S_{E_p}, A_{E_p}, E_{E_p}, s, M_{E_p})$, the features used are:

1. Action label of $\ell$: One boolean for each label in $A_{E_p}$.

2. State labels: One boolean for each label in $A_{E_p}$, indicating the labels of the transitions that have been explored and arrive at $s$. This is a proxy for describing $s$, since state identifiers cannot be used because plant states vary in different instances of the same problem, as the states in individual components and the number of components change.

3. Controllable action: Whether $\ell \in A_{E_p}^C$ (one boolean).

4. Marked state: Whether $s$ and $s'$ belong to $M_{E_p}$ (two booleans).

5. Phases: Whether (at some point in the episode) a marked state has been found, a winning state has been set, and a cycle containing a marked state was closed (three booleans).

6. Child state status: Whether $s'$ is winning, losing, none or not yet explored (four booleans). $s$ should always be none in the exploration frontier, so this is not informed.

7. Uncontrollable transitions: Whether $s$ and $s'$ have uncontrollable transitions and whether they have been explored.

8. Already explored: Whether a transition from $s$ or $s'$ has already been explored (four booleans).

9. Last expanded: Whether $s$ and $s'$ are the last state from which we have expanded from or to (two booleans).

A problem with features 1 and 2 is that the set $A_{E_p}$ usually depends on the instance size $p$. For example, in the Air Traffic problem of the benchmark used (Ciolek et al., 2019), there is one action label $\text{land}_i$ for each plane $i$, and the number of planes increases with $p$. Those indexes need to be removed to address requirement 2. In the example, we would only have one label $\text{land}$ for feature calculation. This is a significant constraint in our learned heuristics, since they cannot disambiguate different components of the same type.

### 5 Experimental Evaluation

In this section we present empirical results for our approach. We report on an implementation of OTF-DCS extended for feature calculation within the MTSA tool (D’Ippolito et al., 2008). The training procedure, which wraps the synthesis algorithm as an RL environment, is available here1. Experiments were run on an Intel i7-7700 CPU with 16GB of RAM and with no GPU. We compare the results with a heuristic that always chooses a random transition in the frontier and with the Ready Abstraction (RA) heuristic (Ciolek et al., accepted 2022).

Heuristics are tested using a benchmark introduced in Ciolek et al. (2019). It contains six parametric control problems: Air Traffic (AT), Bidding Workflow (BW), Travel Agency (TA), Transfer Line (TL), Dinning Philosophers (DP) and Cat and Mouse (CM). All the problems scale in two dimensions, the number of intervening components (parameter $n$) and the number of states per component (parameter $k$).

1https://github.com/tdelgado00/Learning-Synthesis
Abstraction and the green horizontal lines show the mean and max performances of a random policy over 100 executions.

Table 1: Number of transitions expanded by the different approaches on the over 5 random seeds. Standard deviation is shown when it is not zero. Results for the heuristics necessarily the best model found for the instances, unseen during training. RL results correspond to the model selected during the generalization testing step, which is not individually shown with the thinner lines. Learning curves are smoothed using buckets of 5000 steps and a moving average of 10 to improve readability. The red horizontal line shows the (deterministic) performance of the Ready Abstraction and the green horizontal lines show the mean and max performances of a random policy over 100 executions.

|               | (2, 2)       | (2, 3)       | (3, 2)       | (3, 3)       | (4, 4)       |
|---------------|--------------|--------------|--------------|--------------|--------------|
| AT            | 41.8 ± 8.13  | 49.2 ± 16.63 | 109.8 ± 81.25| 267.2 ± 107.0| 1980.6 ± 1290.32 |
| RA            | 55.0         | 52.0         | 227.0        | 483.2 ± 13.03 | 4990.6 ± 138.82 |
| Random        | 53.2 ± 3.66  | 86.6 ± 2.58  | 148.8 ± 28.98| 483.2 ± 13.03 | 4990.6 ± 138.82 |
| Total         | 91.0         | 129.0        | 733.0        | 1279.0       | 21097.0      |
| BW            | 20.0 ± 0.0   | 26.0 ± 0.0   | 44.6 ± 2.58  | 60.8 ± 1.6  | 150.2 ± 7.76  |
| RA            | 66.0         | 102.0        | 525.0        | 1284.0       | 37556.0      |
| Random        | 65.4 ± 7.23  | 146.8 ± 6.91 | 603.4 ± 50.04| 1463.6 ± 146.62 | NA |
| Total         | 132.0        | 230.0        | 997.0        | 2367.0       | 58598.0      |
| CM            | 1258.8 ± 537.43 | 5151.8 ± 2902.42 | 20460.6 ± 10462.67 | NA | NA |
| RA            | 455.0        | 2365.0       | 9764.0       | NA           | NA           |
| Random        | 2352.4 ± 21.7| 10915.2 ± 132.68| 42113.4 ± 99.88 | NA | NA |
| Total         | 5044.0       | 24784.0      | 123754.0     | 1406590.0   | NA           |
| DP            | 54.6 ± 8.11  | 62.4 ± 10.56 | 250.0 ± 127.8 | 263.0 ± 127.76 | 12232.2 ± 593.14 |
| RA            | 32.0         | 35.0         | 81.0         | 87.0         | 202.0        |
| Random        | 224.6 ± 35.47| 268.8 ± 29.0 | 3466.0 ± 480.47| 5056.4 ± 368.53 | NA |
| Total         | 335.0        | 446.0        | 5018.0       | 7671.0       | 212217.0     |
| TA            | 79.8 ± 17.84 | 89.6 ± 17.37 | 284.0 ± 138.5| 303.2 ± 140.37| 1202.4 ± 835.86 |
| RA            | 160.0        | 173.0        | 1074.0       | 2053.0       | 6292.0       |
| Random        | 263.0 ± 16.61| 315.8 ± 5.6  | 2774.6 ± 19.6| 3223.8 ± 45.57| 30403.8 ± 135.58 |
| Total         | 344.0        | 388.0        | 3096.0       | 3568.0       | 31976.0      |
| TL            | 13.4 ± 1.2   | 13.4 ± 1.2   | 17.0 ± 1.6  | 17.0 ± 1.6  | 20.6 ± 2.15  |
| RA            | 10           | 10           | 12           | 12           | 14           |
| Random        | 173.8 ± 115.63| 550.8 ± 405.13| 1434.4 ± 498.88| 3554.0 ± 2081.19 | NA |
| Total         | 1259.0       | 4338.0       | 15296.0      | 94190.0      | NA           |

Table 2: Number of instances solved by the RL, RA and Random heuristics for n and k up to 15, with 10 minutes time-out. RL results correspond to the best models of 5 different random seeds of the training procedure. The random heuristic is averaged over 5 random seeds. Standard deviation is shown when it is not zero.
For each domain we train in the \((n = 2, k = 2)\) instance until no better performance is achieved for the last third of the training steps, for a minimum of 500000 steps. While training, we save the weights of the neural network every 5000 steps and a uniform sample of 100 policies is tested with all values of \(n\) and \(k\) up to 15 with a 5 seconds time-out, only testing instances \((n, k)\) for which both \((n - 1, k)\) and \((k - 1, n)\) have been solved in time. After that, we select the neural network that maximizes the number of instances solved, breaking ties with the minimum sum of expanded transitions. The \((2, 2)\) instances range from 91 total transitions in AT to 5044 in CM, so the number of episodes solved can vary significantly.

The neural network architecture used is a multilayer perceptron with one hidden layer of 20 neurons and ReLU activation. Informal experiments showed no improvement using deeper or wider networks, but they might be useful with a larger set of features that allows more complex policies. Nevertheless, since the focus is on generalization, models should be as simple as possible. The optimizer used was stochastic gradient descent with a constant learning rate of \(1e-5\) and weight decay \(1e-4\). The rate of exploration \((\varepsilon)\) was decayed linearly from 1.0 to 0.01 in the first 250000 steps of training.

Experiments aim to answer the following research questions: (Q1) Do our agents learn? (Q2) Are the learned heuristics competitive in the training instances? (Q3) Do the learned heuristics generalize to larger instances? (Q4) Is the RL approach competitive end-to-end?

Partial observability, sparse rewards and the deadly triad are individually sufficient reasons for learning to fail completely, so whether performance will increase over time is not obvious. Figure 1 shows the evolution of the accumulated reward during training and the performance of the random policy for each problem, answering question (Q1). Learning consistently achieves non-random average performances, with no signs of divergence. The learning curves show consistent improvement for AT, BW and TL. In DP and TA agents seem to achieve a peek performance that is then lost. In CM, the problem with the largest episodes, we only see a slight improvement with respect to the random policy.

Even if learning converges, it is initially unclear whether the features chosen are informative enough to encode good policies and whether the agents will be able to find those policies. The red horizontal lines in figure 1 show the performance of the RA heuristic, answering question (Q2). Our agents rapidly outperform RA in AT, BW, and TA, and the mean performance in DP and TL approximates that of RA quite closely. Learning in CM proves to be challenging for our RL agents, which stay far from the performance of RA.

As it was discussed in section 4.3, good performances in the training instances do not necessarily translate to larger instances. For (Q3), we evaluate the generalization capabilities of our algorithm by comparing the reward obtained by the agents in the testing instances to that obtained by the random policy. Table 1 shows the transitions expanded by the models selected in the generalization testing step in a subset of the testing instances. Our agents perform significantly better than random in all domains, considerably lowering the growth rate of the explored portion of the state space. Furthermore, in AT, BW, and TA, the problems in which training performances were better than RA, expanded transitions were significantly better in the testing instances too, while this was not the case in the problems in which our agents did not surpass RA during training.

As mentioned in section 4.2, our approach entails the overhead of computing features and evaluating the model for every transition in the exploration frontier at each time step, which could be problematic in the end-to-end objective of pushing the frontier of solvable instances within a time budget. Nevertheless, RA also has the significant overhead of estimating distances to marked states for each new state encountered, which requires building a dependency graph. Table 2 shows the number of instances solved on average by the selected models and the baselines, answering question (Q4). Our approach solves significantly more instances than RA in three of the six problems (AT, BW, and TA), shows no difference with RA in two (CM and TL) and solves fewer instances than RA in DP. Finally, across problems, the total number of instances solved was significantly higher than that of RA.

### 6 Discussion and future work

In this work we showed that for the problem of on-the-fly Directed Controller Synthesis it is possible to learn heuristics from scratch while leveraging only what can be learned in very small versions of a problem.

However, as a first iteration of the idea, there is a lot of room for improvements and future work. First, to keep pushing the frontier of solvable instances, it might be necessary to start learning in slightly larger instances, possibly in an incremental manner. However, this seems challenging due to the intensification of the sparse-reward problem for such instances, and because longer episodes have larger accumulated rewards, making it hard for one value function to learn in different instances simultaneously.

Another idea that might be valuable to explore is exploiting the existing heuristic, as proposed by Gehring et al. (2021) for the case of planning, to facilitate learning in tasks that have proven to be challenging for our agents. Using reward shaping in our case is hard because the RA heuristic outputs a list of tuples instead of a real number, but another interesting idea would be to guide the exploration towards the actions selected by RA at the beginning of training.

A third important aspect to improve would be to better address the partial observability of our task, as suggested by Hausknecht and Stone (2015). Our feature vector losses a significant amount of information of the history and general state of exploration, and using a recurrent neural network could allow agents to remember the important aspects of the explored plant and the actions that have been taken. This could be particularly useful in tasks such as DP, where reachability is the main difficulty and a very long strategy needs to be learned to find the marked state. Nevertheless, it is not clear how it can be adapted to our learning algorithm, were all actions are evaluated individually at every step.
References

Bagherzadeh, M.; Kahani, N.; and Briand, L. C. 2020. Reinforcement learning for test case prioritization. *CoRR* abs/2011.01834.

Behjati, R.; Sirjani, M.; and Nili Ahmadian, M. 2010. Bounded rational search for on-the-fly model checking of LTL properties. In Arbab, F., and Sirjani, M., eds., *Fundamentals of Software Engineering*, 292–307. Berlin, Heidelberg: Springer Berlin Heidelberg.

Cassandras, C. G., and Lafortune, S. 2006. *Introduction to Discrete Event Systems*. Springer-Verlag.

Ciolek, D. A.; Braberman, V.; D’Ippolito, N.; Sardiña, S.; and Uchitel, S. 2019. Compositional supervisory control via reactive synthesis and automated planning. *IEEE Transactions on Automatic Control*.

Ciolek, D.; Duran, M.; Zanollo, F.; Pazos, N.; Braier, J.; Braberman, V.; D’Ippolito, N.; and Uchitel, S. (accepted) 2022. On-the-fly Informed Search of Non-blocking Directed Controllers. *Automatica*.

D’Ippolito, N.; Fischbein, D.; Chechik, M.; and Uchitel, S. 2008. MTSA: The Modal Transition System Analyser. In *Proc. of the Int. Conf. on Automated Software Eng.*, ASE.

Ehlers, R.; Lafortune, S.; Tripakis, S.; and Vardi, M. 2016. Supervisory control and reactive synthesis: a comparative introduction. *Discrete Event Dynamic Systems*.

Gehring, C.; Asai, M.; Chitnis, R.; Silver, T.; Kaelbling, L. P.; Sohrabi, S.; and Katz, M. 2021. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. *CoRR* abs/2109.14830.

Hausknecht, M. J., and Stone, P. 2015. Deep recurrent q-learning for partially observable mdps. *CoRR* abs/1507.06527.

Huang, J., and Kumar, R. 2008. Directed Control of Discrete Event Systems for Safety and Nonblocking. *IEEE Trans. Automation Science & Engineering* 5.

Kirk, R.; Zhang, A.; Grefenstette, E.; and Rocktäschel, T. 2022. A survey of generalisation in deep reinforcement learning. *CoRR* abs/2112.01815.

Reddy, S.; Lemieux, C.; Padhye, R.; and Sen, K. 2020. On-the-fly Informed Search of Non-blocking Directed Controllers. *Automatica*.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR* abs/1712.01815.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book.

Watkins, C. J. C. H., and Dayan, P. 1992. Technical note: Q-learning. *Mach. Learn.* 8(3–4):279–292.

Wonham, W. M., and Ramadge, P. J. 1987. On the Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization* 25(3).

Zheng, Y.; Liu, Y.; Xie, X.; Liu, Y.; Ma, L.; Hao, J.; and Liu, Y. 2021. On-the-fly Informed Search of Non-blocking Controllers. *International Conference on Software Engineering (ICSE)*, 423–435.