2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Progress Constraints for Reinforcement Learning in Behavior Trees

Finn Rietz[1,†], Mart Kartašev[2,†], Petter Ögren[2], Johannes A. Stork[1]

***Abstract*—Behavior Trees (BTs) provide a structured and reactive framework for decision-making, commonly used to switch between sub-controllers based on environmental conditions. Reinforcement Learning (RL), on the other hand, can learn near-optimal controllers but sometimes struggles with sparse rewards, safe exploration, and long-horizon credit assignment. Combining BTs with RL has the potential for mutual benefit: a BT design encodes structured domain knowledge that can simplify RL training, while RL enables automatic learning of the controllers within BTs. However, naïve integration of BTs and RL can lead to some controllers counteracting other controllers, possibly undoing previously achieved subgoals, thereby degrading the overall performance. To address this, we propose *progress constraints*, a novel mechanism where feasibility estimators constrain the allowed action set based on theoretical BT convergence results. Empirical evaluations in a 2D proof-of-concept and a high-fidelity warehouse environment demonstrate improved performance, sample efficiency, and constraint satisfaction, compared to prior methods of BT-RL integration.**

I. INTRODUCTION AND MOTIVATION

BEHAVIOR TREES (BTs) provide a reactive plan execution framework that switches between multiple controllers based on the current state, and are commonly used in robotics [1] and video games [2, 3]. Traditionally, such trees are built by a human designer with expert knowledge of the problem, who specifies both the task-switching tree and the controllers in the leaf nodes of the tree (π_1, π_2, π_3 in Fig. 1). Reinforcement Learning (RL), on the other hand, can learn near-optimal controllers [4] for various domains [5, 6] – but suffers from a number of practical limitations: It is often difficult to design scalar-valued reward functions that induce the desired behavior [7, 8] and (safe) exploration in high-dimensional, sparse-reward tasks.

There exists mutual benefit in combining these two frameworks: From the BT perspective, RL can provide a way for learning near-optimal controllers to be used in the BT, while from the RL perspective, BTs can add structured domain knowledge and simplify the learning problem by breaking it down into smaller subproblems that can be learned separately. However, naïvely learning controllers with RL as part of BTs can lead to poor performance. Shortcomings can take the form of internal control switching oscillations [10] or suboptimal handover between controllers [11]. These issues stem from RL algorithms greedily optimizing rewards without considering the constraints imposed by the BT.

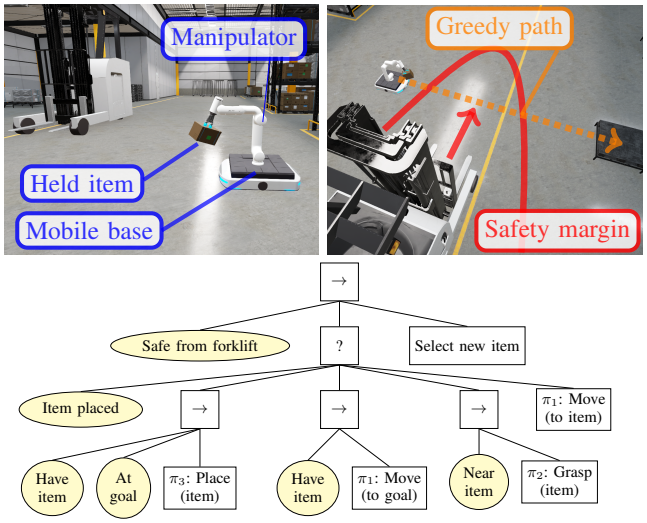

Figure 1: **Top**: High-fidelity warehouse environment. The agent, a small mobile manipulator, must collect and deliver items while avoiding collisions with the bigger, dynamic forklift with unknown dynamics. **Bottom**: Corresponding Behavior Tree with RL controllers π_1, π_2, and π_3. Naïvely learning the controllers with RL can result in unsafe or greedy behavior, while our method accounts for long-term BT progress constraints. Note that the lower part of the BT applies the Implicit Sequence design principle [9].

We suggest that these constraints are automatically identified from the BT, and then taken into account in the RL training. For example, the BT in Fig. 1 works best if the `Place` controller π_3 remains close to the goal location, and does not prematurely drop the item, since if it moves away `Move` has to move it back, and if it drops the item in the wrong place, `Grasp` has to pick it up again. Note that the `Move` controller π_1 appears twice in the BT, and is subject to different constraints, depending on the state. When moving towards the item, no effort is needed to avoid dropping it.

Thus, in this paper, we investigate how to integrate RL with BTs to leverage the structured domain knowledge that BTs provide while ensuring that the RL learns controllers that respect the constraints imposed by the BT. Our approach introduces so-called *progress constraints*, which prevent RL controllers from undoing already completed steps in the BT task, such as dropping an object that was just grasped. We exploit theoretical convergence results for BTs to identify the constraint set [12], and then use feasibility estimation to learn estimators for these constraints.

[†]Equal contribution

[1]Adaptive and Interpretable Learning Systems Lab, Department of Computer Science, Örebro University.
[2]Robotics, Perception and Learning Lab, School of Electrical Engineering and Computer Science, Royal Institute of Technology.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Thus, we present the first method for learning both RL controllers and constraints within BTs and make the following contributions:

1) An extension of BT convergence concepts previously used for RL in BTs [10] using the results in [12] to address a broader class of BTs.
2) Progress constraints: We show how to learn feasibility estimators that constrain the action set to avoid undoing the progress of BTs.
3) We show how to use these estimators in RL algorithms, thereby providing a general-purpose learning algorithm for BTs with RL controllers.
4) We open-source the warehouse environment at https://github.com/martkartasev/CBTRL-Warehouse

II. RELATED WORK

One of the earliest works on combining BTs and RL is [13], which introduces a "learning node" that applies naïve Q-learning [14], without accounting for the implicit constraints a BT imposes on its controllers discussed in this paper. Prior work [10] and our results show that this approach does not respect the constraints induced by the BT, which can be attributed to the greedy, off-policy backup in Q-learning [15].

The work that is most similar to this one is [10]. Both papers are based on the idea of using convergence analysis of BTs to improve RL when learning controllers. There are however two fundamental differences. First, [10] uses convergence results from [16] that only cover a specific class of BTs (backward chained designs) whereas this paper uses convergence results from [12] that apply to all BTs. Second, in [10] reward shaping is used, giving undesired transitions a large negative reward, whereas in this paper we construct progress constraints using feasibility estimators to avoid such transition all-together, independently of the reward functions. In the evaluation we compare the results to that of [10].

Another recent work combining BT and RL [11] deals with the optimization of control switching on the boundary between two controllers. While dealing with a similar setting as this paper, [11] addresses a different issue that aims to improve the handover between different controllers by giving rewards to one controller based on the preferences of the next.

A review of safe control can be found in [17]. We draw motivation from methods that identify safe state-space regions that controllers should not leave, i.e. keep invariant. In particular, we make use of dynamic programming "feasibility" methods, as introduced in [18] and expanded on in [19], that are traditionally used for identifying sets of safe states. Differently from these prior works, we are applying feasibility methods not for safe control but to track and support the progress in a BT and use the resulting estimators for constraining the RL controllers. A comprehensive review of constrained RL can be found in [20], but that work contains no results on how BTs can be used to identify contraints. Finally, we apply "action masking" methods as described in [21, 22, 23]. However, none of those works share our BT application.

III. BACKGROUND

We begin with a summary of relevant background information. First, we give a formal definition of BTs in Sec. III-A and review important convergence properties of BTs in Sec. III-B and provide the necessary definitions from RL in Sec. III-C and lastly review feasibility analysis in Sec. III-D.

A. Behavior Trees

Let the state be $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^n$ and evolve according to the (unknown) discrete-time dynamics $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u})$. With control actions $\mathbf{u} \in \mathcal{U} \subset \mathbb{R}^k$, the policy $\pi_i : \mathcal{X} \to \mathcal{U}$ is the controller that runs when the BT node i is executing [12].

Definition 1. *(Behavior Tree) A BT is a two-tuple $\mathcal{T}_i = \{\pi_i, r_i\}$, where $i \in \mathbb{N}$ is the index of the tree, π_i is the controller and $r_i : \mathbb{R}^n \to \{\mathcal{R}, \mathcal{S}, \mathcal{F}\}$ is the return status that can output either* Running *($\mathcal{R}$),* Success *($\mathcal{S}$), or* Failure *($\mathcal{F}$). The return status partitions the state space into the Running region $R_i = \{x : r_i(x) = \mathcal{R}\}$, Success region $S_i = \{x : r_i(x) = \mathcal{S}\}$ and Failure region $F_i = \{x : r_i(x) = \mathcal{F}\}$.*

The return status r_i is used when combining BTs into larger trees. While every node in the tree can formally be considered an individual tree, we often call certain types of nodes by different names [24]. Nodes that execute controllers are called *behaviors* (rectangles containing π_0, π_1, π_2 in Fig. 1). Nodes that return only Success *(true)* or Failure (*false*) are known as *conditions* that can be used as predicates (ellipsoids in Fig. 1).

For example, consider the BT in Fig. 1. Nodes are iterated from left to right. The Sequence ($\rightarrow$) node returns `Success` if all children succeed, while the Fallback (?) node returns `Success` if any child succeeds. The fallback node in the figure represents a so-called implicit sequence [9], with intended progression from right to left. If the condition `Safe` (from Forklift) is unsatisfied (returns `Failure`), the whole tree returns `Failure` and an emergency stop is triggered. The middle subtree is executed only if the condition `Safe` is satisfied and `Item Placed` is not. If conditions like `At Goal` or `Have Item` change their values during execution of controller π_i, the BT switches to the appropriate controller. For instance, if `Have Item` becomes unsatisfied while executing π_3 :`Place` or π_1 : `Move (to goal)`, the BT will return to executing π_2 in the next time step.

B. Behavior Tree convergence analysis

We build on the BT convergence analysis [16, 12] to constrain the RL controllers in a BT. The key idea is to make sure the RL controllers keep the so-called *convergence sets* invariant while striving to complete each subtask. In order to define these sets, we first have to define so-called influence and operating regions. The influence regions can be recursively computed considering the BT structure.

Definition 2. *(Influence region) For a given node i, we define $p(i)$ as the parent node of i and $b(i)$ as the closest brother/sibling (having the same parent) node to the left of i. Based on that, the Influence region I_i for node i can be determined recursively as:*

$$
\begin{aligned}
I_i &= \mathcal{X} && \text{If } i \text{ is the root} \\
I_i &= I_{p(i)} && \text{If } \nexists b(i) \ \ \& \ \ \exists p(i) \\
I_i &= I_{b(i)} \cap S_{b(i)} && \text{If } p(i) \text{ is a Sequence } \exists b(i) \\
I_i &= I_{b(i)} \cap F_{b(i)} && \text{If } p(i) \text{ is a Fallback } \exists b(i)
\end{aligned}
$$

Intuitively, the influence region I_i is the part of the state-space in which subtree i is able to influence the execution of the BT, by either its controller π or its return status r_i.

Based on influence regions, we can define operating regions, which are the parts of the space where node i has return status `Running` and executing its controller π_i.

Definition 3. *(Operating Regions) The operating region $\Omega_i = I_i \cap R_i$ of node i, is the intersection of it's influence region I_i and running regions R_i. For a given node i, with operating region Ω_i, the corresponding controller π_i is executed when $\mathbf{x} \in \Omega_i$. The operating regions of the children of i are a partitioning of Ω_i.*

Thus, a given BT induces a partition of the state space into operating regions Ω_i, where controller π_i is executing when the state $\mathbf{x} \in \Omega_i$, for more details, see [12]. Given these concepts, we can state the main convergence result from [12], giving sufficient conditions for reaching the overall success region S_0, corresponding to completion of all tasks.

Theorem 1. *(Convergence of BTs, from [12]) Given the set of BT leaf nodes $J = \{1, ..., j\}$, let*

$$C_i = ((\bigcup_{j \geq i} \Omega_j) \cup S_0). \tag{1}$$

If there exists a re-labeling of the nodes such that for all $i \in J$, the convergence set C_i is invariant under π_i and there exists a time-horizon h, such that if $\mathbf{x}_t \in \Omega_i$ then $\mathbf{x}_{t+h} \notin \Omega_i$. Then, the state $\mathbf{x}$ will enter the success region S_0 of the root of the tree (the overall success of the BT) in finite time.

The intuition behind Theorem 1 is as follows: Assume some intended progression (see Remark 1) through the operating regions $\Omega_1, \ldots \Omega_J$ and let us label the leaf nodes of the BT such that the controllers for the operating regions are numbered in order of increasing intended progression. Then, if we can guarantee that all controllers strictly transition between operating regions according to $\Omega_j \rightsquigarrow \Omega_{j+k}, k \geq 1$, as illustrated in Fig. 2, each C_i is invariant under the respective π_i and the BT is guaranteed to reach a state $\mathbf{x} \in S_0$ in finite time. Conversely, if any controller violates this order, it is undoing progression that has already been achieved, forcing the agent to go back and do the same task again. The key idea of our method is make use of the invariance constraints from Theorem 1 on C_i and imposing them on the RL controller π_i.

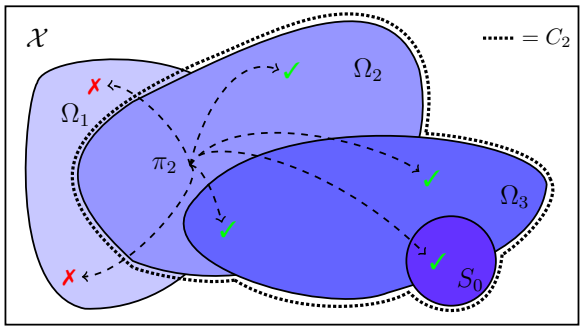

Figure 2: Visualization of operating regions, a controller, and its convergence set. The controller π_2 may leave its operating region Ω_2 but must remain in its convergence set C_2.

Remark 1. *(Intended order of progression) The intended order of progression is often clear for a manually designed BT. If the BT was designed using the backward chaining approach [25], the intended order of progression can be found by a depths/left first numbering of the leaves. Another example is the so-called implicit sequence, see [24], and the lower part of the BT in Fig. 1, where the intended progression would be right to left.*

Remark 2. *For the example in Figure 1 we have an intended progression from right to left due to the implicit sequence, i.e. `Move(ToItem)` $\rightarrow$ `Grasp` $\rightarrow$ `Move(ToGoal)` $\rightarrow$ `Place)`, with the corresponding convergence sets to be kept invariant*

$$
\begin{aligned}
C_{Move(ToItem)} =& \Omega_{Move(ToItem)} \cup \Omega_{Grasp} \\
& \cup \Omega_{Move(ToGoal)} \cup \Omega_{Place} \cup S_0 \\
C_{Grasp} =& \Omega_{Grasp} \cup \Omega_{Move(ToGoal)} \cup \Omega_{Place} \cup S_0 \\
C_{Move(ToGoal)} =& \Omega_{Move(ToGoal)} \cup \Omega_{Place} \cup S_0 \\
C_{Place} =& \Omega_{Place} \cup S_0.
\end{aligned}
$$

C. Reinforcement Learning

Reinforcement learning problems are formalized as Markov Decision Processes (MDPs). An MDP is a tuple $\mathcal{M} \equiv \langle \mathcal{X}, \mathcal{U}, r, f, \gamma, \mu \rangle$, where $\mathcal{X}$, $\mathcal{U}$, and f are aligned with the BT notation above and respectively denote the state-space, action-space, and transition function. $r : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}$ is a scalar-valued reward function, $\gamma \in [0, 1]$ is a discount factor, and μ is a distribution over initial states. The goal in RL is to find a policy (henceforth referred *controller*) $\pi : \mathcal{X} \times \mathcal{U} \rightarrow [0, 1]$ that maximizes the return

$$\mathbb{E}_{(\tau \sim \pi)} \Big[\sum_{t=0}^{\infty} \gamma^t r(\mathbf{x}_t, \mathbf{u}_t) \Big], \tag{2}$$

where $\mathbf{x}_t \in \mathcal{X}$, $\mathbf{u}_t \in \mathcal{U}$, and $(\tau \sim \pi)$ is shorthand for denoting trajectories τ with actions sampled according to the policy and states sampled according to the transition dynamics.

D. Feasibility Estimation

The main idea in feasibility estimation is to use dynamic programming techniques for safety analysis, as suggested in [18]. Imagine we have some set of *failure states* that we want to avoid because they either correspond to collisions or are undesirable in some other way. Then we call the complement of these states the *constraint set* $\mathcal{K} \subset \mathcal{X}$. Assuming $\mathcal{K}$ is closed, let $l : \mathcal{X} \to \mathbb{R}$ such that $l(\mathbf{x}) > 0 \Leftrightarrow \mathbf{x} \in \mathcal{K}$ (for example, $l(\mathbf{x})$ might be a measure of the distance from $\mathbf{x}$ to the nearest failure state). We can then define the state feasibility function as

$$V(\mathbf{x}) = \sup_{\pi} \inf_{t \geq 0} l(x(t)), \tag{3}$$

where $x(t)$ is a state trajectory resulting from the controller π. Intuitively, $V(\mathbf{x})$ measures how close to the set of failure states we will come in the future, starting from $\mathbf{x}$ and applying a controller that optimally avoids the unsafe set. It is shown in [18] that the state feasibility function satisfies the discrete-time Bellman equation

$$V(\mathbf{x}) = \min[l(\mathbf{x}), \max_{\mathbf{u}} V(f(\mathbf{x}, \mathbf{u}))]. \tag{4}$$

However, this equation does not induce a contraction, which is needed to ensure convergence. To remedy this, Fisac et al. [18] introduce the following estimator

$$V(\mathbf{x}) = (1-\gamma) l(\mathbf{x}) + \gamma \min[l(\mathbf{x}), \max_{\mathbf{u}} V(f(\mathbf{x}, \mathbf{u}))], \tag{5}$$

which induces a contraction mapping and show that the solution converges to Eq. (4) as $\gamma \to 1$. Finally, the state-action feasibility function Q can be found by solving

$$Q(\mathbf{x}, \mathbf{u}) = (1-\gamma) l(\mathbf{x}) + \gamma \min[l(\mathbf{x}), \overbrace{\max_{\mathbf{u}'} Q(f(\mathbf{x}, \mathbf{u}), \mathbf{u}')}^{V(\mathbf{x}')}]. \tag{6}$$

Now, for any state $\mathbf{x} : V(\mathbf{x}) > 0$, there must be a non-empty set of safe actions

$$\mathcal{U}(\mathbf{x}) = \{\mathbf{u} : Q(\mathbf{x}, \mathbf{u}) \geq 0\}, \tag{7}$$

and as long as we select $\mathbf{u}_t \in \mathcal{U}(\mathbf{x})$, the next state will also be in $\mathcal{K}$.

IV. Method: Feasibility-constrained Behavior Tree Reinforcement Learning

A. Problem definition

We now present our method for solving *Progress-Constrained Behavior Tree Reinforcement Learning* (CBTRL) problems: Given a BT $\mathcal{T}_0$ with J behavior nodes that partitions the state $\mathcal{X}$ into operating regions $\Omega_1, \ldots, \Omega_J$, we wish to use RL to learn controllers $\pi_1, \ldots, \pi_I$, $I \leq J$, such that the overall task modeled by the BT is solved successfully when the learned controllers are used in the BT's behavior nodes. Since we often want to reuse the same controller π_i in multiple behavior nodes, e.g. the `Move` controller in Fig. 1, we allow $I \leq J$ and denote the set of behavior nodes in which controller i is used by $\mathcal{J}_i$. For each controller i that we want to learn, we define a "subtask" MDP $\mathcal{M}_i = \langle \mathcal{X}, \mathcal{U}, r_i, f, \gamma, \mu_i \rangle, i \in \{1, \ldots I\}$ that shares the BTs state space $\mathcal{X}$, action space $\mathcal{U}$, and transition function f. The distribution of initial states, μ_i, is a subset of the operating region Ω_i and can be induced with the controllers in $\mathcal{T}_{i-1}$, while each reward function r_i can be chosen to facilitate quick learning of controller π_i to reach the respective success region $\mathcal{S}_i$. However, as alluded to before, directly maximizing the different $\mathcal{M}_i$ with RL does not necessarily induce convergent behavior of the overall system [10].

B. Implementing Progress constraints

To solve CBTRL problems, we take the convergence analysis of BTs into account. According to Theorem 1, the BT is guaranteed to converge to the overall success region S_0 if each controller π_i leaves its operating region Ω_i in finite time while keeping the convergence region C_i invariant. With this in mind, let Π_i denote the set of controllers under which C_i remains invariant. While computing Π_i directly is generally intractable, Equations (6, 7) provide a method to estimate the set of feasible actions $\mathcal{U}(\mathbf{x})$ that preserve invariance of a constraint set $\mathcal{K}$.

Therefore, for controllers used in a single behavior node, we directly set $\mathcal{K} = C_i$ and restrict the action space of $\mathcal{M}_i$ according to $\mathcal{U}_i(\mathbf{x_t})$, effectively projecting π_i into Π_i. For controllers used in multiple behavior nodes, the applicable constraint set depends on which behavior node $j \in \mathcal{J}_i$ selected the controller. This can be uniquely determined by identifying the active operating region $\Omega_j \subset C_i$ for the current state $\mathbf{x}$. Since convergence regions are nested, i.e., $C_j \subseteq C_i$ for any $j \geq i$, masking the action space according to $\mathcal{U}_j(\mathbf{x})$ still ensures that C_i remains invariant. Thus, by selecting the appropriate constraint set C_j based on the active operating region and masking the action space according to $\mathcal{U}_j(\mathbf{x})$, we can use controller π_i in multiple behavior nodes $\mathcal{J}_i$, while still guaranteeing that the corresponding convergence set C_i is kept invariant. This allows us to use any RL algorithm that supports invalid action masking [21, 22, 23] to optimize π_i while satisfying progress constraints. This is the core idea of our method, based on which we derive our full learning algorithm for solving CBTRL problems.

C. Learning algorithm

Our method learns the controllers $\pi_1, \ldots, \pi_I$ iteratively and separately, following the intended order of progression (Remark 1) in the BT $\mathcal{T}$ to ensure that subtrees $\mathcal{T}_0, \ldots, \mathcal{T}_{i-1}$ and controllers are available to induce μ_i when learning controller i. If there are constraints affecting even the first controller π_1, like in Fig. 1, we assume the availability of some dataset of transitions $\mathcal{D}_0$ for training the feasibility estimator Q_1 for C_1. If unavailable, $\mathcal{D}_0$ can be obtained using exploration techniques like [26] – more detailed data collection procedures, methods obtaining Q_1 analytically, or computing Q_1 with access to environment dynamics [27] are applicable but out of scope for this work.

Learning Q_1 requires a labeling function $l_1 : \mathcal{X} \to \mathbb{R}$ such that $l_1(\mathbf{x}) > 0 \Leftrightarrow \mathbf{x} \in \mathcal{C}_1$. We define l_1 as an binary indicator function $l_1 : 1 \iff \mathbf{x} \in C_1$, which we can compute using Theorem 1 by checking whether $\mathbf{x} \in (\bigcup_{j \geq 1} \Omega_j) \cup S_0$. We now compute Q_1 using the transitions in $\mathcal{D}_0$ labeled with l_1, following [18] by performing SGD w.r.t. the parameter ϕ_1 on the Bellman feasibility residual from Equation (6). Generally, for some constraint j, this means minimizing

$$L(\phi_j, \mathcal{D}) = \mathbb{E}_{(\mathbf{x},\mathbf{u},\mathbf{x}')\sim\mathcal{D}}\Big[\big(\bar{y} - Q_j(\mathbf{x}, \mathbf{u}, \phi_j)\big)^2\Big],$$
$$\bar{y} = (1-\gamma) l_j(\mathbf{s}) + \gamma \min\Big(l_j(\mathbf{x}), \max_{\mathbf{u}'} Q_j(\mathbf{x}', \mathbf{u}', \phi_j)\Big). \tag{8}$$

With access to Q_1, we can learn π_1 by optimizing $\mathcal{M}_1$ while masking the action space according to $\mathcal{U}_1$ and storing the collected transitions as $\mathcal{D}_1$.

We follow the same scheme in all subsequent iterations of our method. In iteration j, we first train the feasibility estimator Q_j using all data available thus far, i.e. $\mathcal{D}_j = \mathcal{D}_0 \cup \cdots \cup \mathcal{D}_{j-1}$, then train controller π_j using the action mask $\mathcal{U}_j$ induced from Q_j. If controller π_j is shared with some previous behavior nodes $\mathcal{J}_j = \{l, \ldots, k\} \leq j$, the action space is masked based on Ω_k for each state.

Since the i-th feasibility estimator Q_i and RL policy π_i can be learned independently from prior components, the overall complexity of this approach grows only linearly with I. Lower priority RL controllers with higher indices are subject to increasingly many constraints, which naturally restricts their freedom over actions. This is by design of the priority induced by the BT and does not increase the computational complexity or difficulty of the RL objective, in fact, the smaller action space simplifies exploration.

To summarize, for each RL controller in $\{\pi_1, \ldots \pi_I\}$, we first produce the feasibility estimator for the corresponding convergence region C_i and then learn the RL controller π_i subject to that constraint.

Remark 3. *(Customization) Since each controller π_i in a BT operates within a fixed region Ω_i, we are not restricted to using learning for all controllers. There can be a mix or RL and model based controllers. Furthermore, due to the off-policy nature of the feasibility estimation, if prior data exists or the constraint can be inferred from a model, it can be applied directly without using the iterative process described above.*

V. EXPERIMENTS

A. Experimental setup

Baselines:

- *Standard RL*: A standard RL agent that receives -1 reward for each subtask that is *not* achieved at step t. Since this baseline features no BT, it reveals how the structured domain knowledge from the BT can affect the final agent.

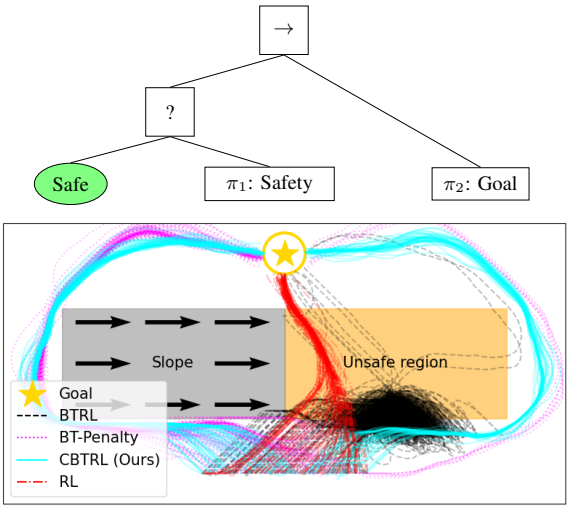

Figure 3: 2D goal-reach environment. **Top**: Task-switching BT with RL-based controllers. The BT selects the `goal` controller π_2, when the state is in its convergence set C_2, which corresponds to states outside of the unsafe region and the slope. **Bottom**: Trajectories generated by different methods. Our method converges to a near-optimal behavior for the BT task of reaching the goal without entering the unsafe region. Note that with our method, π_2 never violates the progress constraint "safe" during learning.

- *BTRL (DQN / PPO)*: Proposed by [13] as a Learning Action Node. This method employs standard, unconstrained RL principles to learn the controllers in the BT, using the environment subtask rewards, and can be seen as an ablation of our method that does not use progress constraints.
- *BT-Penalty*: A reward penalty is added to transitions that violate progress constraints when learning the controllers in the BT with RL. This is similar to the method used in [10] but without presuming a positive reward structure for the task reward.

2D goal-reach environment:

This environment, shown in Fig. 3, features a point-mass agent that has to navigate to a fixed goal location while avoiding an unsafe region. The state-space is $\mathbb{R}^4$ and contains the agent's x and y position and velocity. The discrete action space is of size 25 and corresponds to accelerations at differing angles and magnitudes. Episodes are truncated after 200 steps. When the agent is on the slope, actions have no effect, simulating the agent sliding into the unsafe region. The slope serves to illustrate the need for temporally extended feasibility estimation and progress constraints, since stepping onto the slope results in an inevitable but delayed constraint violation. While knowledge about the unsafe region is encoded in the BT, due to the model-free setting, the agent must learn about the dynamics of the slope by exploration. The subtask reward functions for which we learn controllers are as follows:

- r_1 Safe: -1 reward for each step spent in the unsafe area,

otherwise 0.

- r_2 Goal: A negative term inversely proportional to the distance from the goal.

Warehouse environment:

To test the method's scalability to more complex environments, we implemented the example seen in Fig. 1. Here, the agent is a mobile manipulator that has to pick up and deliver items while avoiding collision with a dynamic forklift, for which the dynamics are unknown. The environment features an observation space of 37 continuous vector features containing the position of the manipulator's joints in the body frame, the state of the suction cup, the pose and velocity of the agent and forklift, as well as the varying positions of the box and goal locations in the body frame. The action space of 6250 discrete actions represents a product space of the possible controls for the wheels, arms, and suction cup of the agent. The subtask policies `Move`, `Grasp`, and `Place` have the following reward functions.

- r_1 Move: A positive term proportional to the decrease in the agent's distance to the goal and a negative term for colliding with the static environment (walls, shelves).
- r_2 Grasp: A positive term proportional to the decrease of the distance of the arm to the item and a second term proportional to the decrease of the item's distance to the desired grasping position.
- r_3 Place: A positive term proportional to decreasing the box's distance to the placing position.

The MDPs also use a small negative penalty for passing time.

This environment only uses the BTRL, BT-Penalty, and CBTRL methods, as the Standard RL policy did not converge to a solution within the project's computational constraints. To demonstrate the effect of the constraint, we also test applying the constraint to the BTRL and BT-Penalty methods without training them explicitly with the constraint active, as is done in CBTRL. All methods use PPO to train the policy and store the experiences in a dataset for later use during the training of the feasibility estimators. Each controller is trained separately in a dedicated learning environment to reduce the complexity of the learning process. There is also a dynamic obstacle in the form of the Forklift, which makes the "Safe" condition depend on the changing forklift position.

For evaluation, we created a scenario that requires a combination of all the separately trained controllers in a BT using task settings analogous to the training tasks in a previously unforeseen configuration. The results of this evaluation can be seen in Table I.

B. Results and discussion

We now analyze our proposed method, CBTRL in comparison to the above-mentioned baselines on our two environments. We focus on comparisons with respect to success rate, progress constraint violation, and sample complexity.

a) ***Progress constraints induce RL controllers that successfully solve the BT tasks****:* It can be seen that progress constraints have a strong effect on the agent, both during learning and in the final behavior. For the 2D environment, Fig. 3 shows trajectories of the final agents, while Fig. 4 shows metrics collected during the training of π_2. CBTRL solves the task modeled by the BT much faster than the baselines and, crucially, without violating constraints even during training. Standard RL, without the task-switching BT structure, learns to drive the shorter path through the unsafe region. While the penalty-based method also learns to solve the BT task, it initially incurs constraint violations and learns slower, as the large penalties have a destabilizing effect on the RL training. Lastly, the unconstrained BTRL method [13] does not account for progress constraints and gets stuck in a task-switching loop: Underneath the unsafe region, π_2 attempts to drive upwards towards the goal and enters the unsafe region. Inside the unsafe region, the π_1 safety controller is selected, driving down, out of the unsafe region. Now π_2 again attempts to drive upwards to the goal, and the cycle repeats. This is a typical problem that might occur when doing task-switching.

A similar problem exists in the Warehouse scenario results of Table I. The individually trained BTRL-Move behavior causes *Have Item* violations by dropping the box. This causes the system to chatter between the controllers, making little

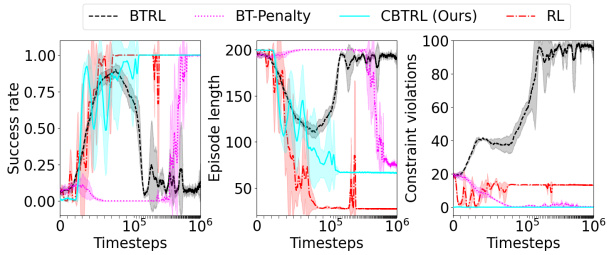

Figure 4: Empirical results on 2D navigation environment. We plot the mean of five random seeds, the shaded area corresponds to one standard deviation around the mean. To enhance readability, the x-axis is linearly scaled in the range $[0, 10^5]$ and logarithmically scaled from 10^5 to 10^6.

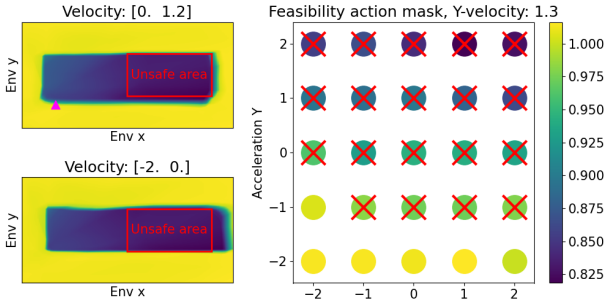

Figure 5: **Left**: Learned state feasibility function for the unsafe region in the 2D environment accounts for both agent and environment dynamics. **Right**: Induced action space mask $\mathcal{U}_1$ with the agent placed at ▲ and velocities as in the top image on the left. Even though the `safety` constraint violation only occurs in the unsafe area, the constraint prevents the agent from stepping onto the slope.

Evaluated Method	Steps to Complete	Success Rate	Timeout Rate	Failure Rate	Have Item Violations # : %	Near Item Violations # : %	At Goal Violations # : %	Safe Violations # : %
BTRL	1186.99 ± 959.33	78%	3%	19%	1211 : 54.55%	0 : 0%	62 : 2.99%	400 : 37.66%
BTRL **w. pc**	2782.40 ± 1842.94	44%	31%	25%	2 : 0.20%	0 : 0%	99 : 1.30%	389 : 34.37%
BT-Penalty	1154.91 ± 624.51	83%	1%	16%	155 : 13.77%	0 : 0%	1 : 0.10%	481 : 43.46%
BT-Penalty **w. pc**	1208.41 ± 500.16	82%	1%	17%	11 : 1.10%	0 : 0%	1 : 0.10%	296 : 27.27%
CBTRL (ours)	$\mathbf{1005.59 \pm 302.97}$	**93%**	**0%**	**7%**	0 : **0%**	0 : 0%	0 : **0%**	198 : **19.30%**

Table I: Agent evaluation in the warehouse environment. Values are given over 1000 random episodes on an unseen but analogous setting. **w. pc** indicates evaluation ***with progress constraint***, where the constraint is applied after training but not during training. Violations given as total number (#) and percentage of episodes that had violations (%).

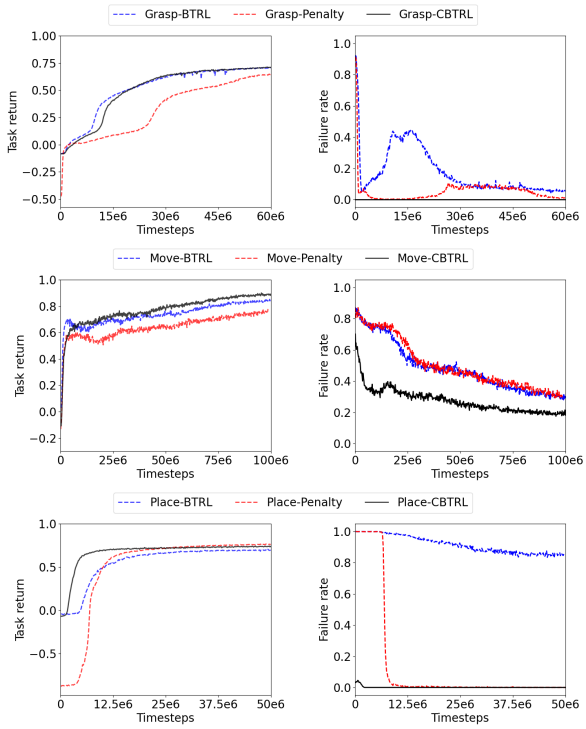

Figure 6: Controller training in the Warehouse environment (`Grasp` (top), `Move` (middle), `Place` (bottom)).

progress. This behavior is reduced in BT-Penalty, which learns to prevent the negative reward from the *Have Item* violation, and is eliminated in the CBTRL method, which prohibits releasing the box. Applying the constraints after training to BTRL and BT-Penalty also eliminates such violations (rows 2 and 4), but induces problems discussed in the next paragraph.

b) ***Ad-hoc constraints degrade performance and introduce failure modes****:* Learning RL controllers decoupled from progress constraint estimators and enforcing the constraints only at inference time might make sense from a modularity perspective, but leads to suboptimal performance and inconsistent behaviors. Table I shows that applying progress constraints to controllers learned in unconstrained fashion (BTRL w.pc and BT-Penalty w.pc) can drastically lower respective constraint violations. However, this comes at the cost of considerably increased timeout rates and reduced overall task completion. This problem stems from the fact that the policy was never trained under the constraint-aware action space. For example, the BTRL-Place controller might learn to complete its subtask by dropping the item from an unsafe height – a behavior that becomes invalid once the progress constraint is enforced. Since the controller never experienced this restriction during training, it fails by exceeding the mission time limit when its preferred behavior is disallowed during inference. The CBTRL method, in contrast, integrates constraints during training, thereby forcing controllers to learn feasible strategies from the outset, leading to consistent and successful task execution. This demonstrates that progress constraints must be incorporated throughout training, not merely applied afterward, in order to achieve the desired behavior within BTs.

c) ***Feasibility estimation captures long-term consequences of actions w.r.t progress constraints****:* The importance of long-term feasibility estimation w.r.t BT progress constraints can be seen in Fig. 5. While the safety constraint violation only occurs when the agent enters the unsafe region, The use of the Bellman feasibility equations also identifies the "slope" area as infeasible, i.e. as outside of the convergence set C_2 for π_2. This is correct since once the agent steps onto the slope, it will inevitably slide into the unsafe area. If we were to mask the action space by forbidding only those actions that immediately violate the progress constraints, stepping onto the slope would be wrongfully allowed. Thus, Fig. 5 shows that feasibility estimation can be used to capture the long-term dynamics w.r.t progress constraints, which is needed to estimate the convergence region C_i for the controllers in the BT.

d) ***BT task decomposition with progress constraints reduces RL sample complexity****:* The structured domain knowledge in a BT simplifies the RL learning problem by limiting the policy search space. The effect can be most prominently seen in Fig. 6 during the learning of the Place behaviors in the Warehouse environment. We see that the constrained *CBTRL-Place* behavior, which is constrained by the estimator trained with data from the *BTRL-Grasp* behavior, converges to a stable result after around 13×10^6 experiences; much faster than *BTRL-Place*, which needs about 22×10^6 experiences. This is because all actions in the product space where the box is released are removed and do not have to be explored. This can be verified by the failure rate of zero during the training of "CBTRL-Place". Similarly, in Fig. 4, CBTRL converges after roughly 10^5 steps, while the BT-penalty method requires roughly 10^6 steps.

e) ***Feasibility estimator overhead****:* We used 5M transitions collected by the BTRL baseline for training the

first estimator for CBTRL-Move. Compared to the 100M transitions for the policy itself, the additional data required is only 5% extra of what is required for the policy. For the BT in Fig. 3, no additional data was needed, since the feasibility estimator for π_2: Goal can be trained using the transition collected during learning of π_1: Safety. The overhead from the feasibility estimators varies by domain and BT, but is generally small compared to RL's high data demands. A detailed analysis of data requirements for feasibility estimation is interesting, but beyond this work's scope.

VI. CONCLUSIONS

The main contribution of this paper is an algorithm for learning controllers in a BT with RL. We propose novel *progress* constraints based on feasibility estimation and BT convergence properties, which are then used to learn the controllers in the BT with constrained RL. A limitation of this work lies in potentially insufficient data collection for learning the progress constraints, since we assume that the data collected during exploration with RL suffices for training robust feasibility estimators. We see a more informed and optimized way of gathering data for learning these estimators, or the integration of model-based progress constraints, as important future work.

REFERENCES

[1] M. Iovino et al. "A survey of behavior trees in robotics and ai". In: *RAS* 154 (2022).

[2] D. Isla. "Handling Complexity in the Halo 2 AI". In: *Proceedings of the Game Developers Conference (GDC)*. 2005.

[3] O. Biggar et al. "On Modularity in Reactive Control Architectures, with an Application to Formal Verification". In: *ACM Transactions on Cyber-Physical Systems (TCPS)* (May 2022).

[4] R. S. Sutton et al. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] J. Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602.7897 (2022), pp. 414–419.

[6] J. Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nat.* 588.7839 (2020), pp. 604–609.

[7] C. Tessler et al. "Reward constrained policy optimization". In: *arXiv preprint arXiv:1805.11074* (2018).

[8] P. Vamplew et al. "Scalar reward is not enough: a response to Silver, Singh, Precup and Sutton (2021)". In: *Auton. Agents Multi Agent Syst.* 36.2 (2022), p. 41.

[9] M. Colledanchise. "Behavior Trees in Robotics". PhD thesis. KTH Royal Institute of Technology, 2017.

[10] M. Kartašev et al. "Improving the Performance of Backward Chained Behavior Trees That Use Reinforcement Learning". In: *IEEE IROS*. 2023.

[11] M. Kartašev et al. "Improving the Performance of Learned Controllers in Behavior Trees Using Value Function Estimates at Switching Boundaries". In: *IEEE Robotics and Automation Letters* (2024).

[12] P. Ögren et al. "Behavior Trees in Robot Control Systems". In: *Annual Review of Control, Robotics, and Autonomous Systems* 5.1 (2022).

[13] R. d. P. Pereira et al. "A framework for constrained and adaptive behavior-based agents". In: *arXiv preprint arXiv:1506.02312* (2015).

[14] C. J. Watkins et al. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[15] S. J. Russell et al. "Q-decomposition for reinforcement learning agents". In: *Proceedings of the 20th international conference on machine learning*. 2003, pp. 656–663.

[16] P. Ögren. "Convergence analysis of hybrid control systems in the form of backward chained behavior trees". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6073–6080.

[17] K.-C. Hsu et al. "The safety filter: A unified view of safety-critical control in autonomous systems". In: *Annual Review of Control, Robotics, and Autonomous Systems* 7 (2023).

[18] J. F. Fisac et al. "Bridging hamilton-jacobi safety analysis and reinforcement learning". In: *2019 International Conference on Robotics and Automation*. IEEE. 2019, pp. 8550–8556.

[19] D. Yu et al. "Reachability constrained reinforcement learning". In: *ICML*. 2022, pp. 25636–25655.

[20] S. Gu et al. "A review of safe reinforcement learning: Methods, theory and applications". In: *arXiv preprint arXiv:2205.10330* (2022).

[21] G. Kalweit et al. "Deep constrained q-learning". In: *arXiv preprint arXiv:2003.09398* (2020).

[22] S. Huang et al. "A closer look at invalid action masking in policy gradient algorithms. arXiv 2020". In: *arXiv preprint arXiv:2006.14171* (2020).

[23] F. Rietz et al. "Prioritized Soft Q-Decomposition for Lexicographic Reinforcement Learning". In: *ICLR*. 2024.

[24] M. Colledanchise et al. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[25] M. Colledanchise et al. "Towards Blended Reactive Planning and Acting Using Behavior Trees". In: *IEEE ICRA*. May 2019.

[26] B. Thananjeyan et al. "Recovery RL: Safe Reinforcement Learning With Learned Recovery Zones". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4915–4922.

[27] A. D. Ames et al. "Control barrier functions: Theory and applications". In: *2019 18th European control conference*. IEEE. 2019, pp. 3420–3431.