Safe Explainable Agents for Autonomous Navigation using Evolving Behavior Trees

Nicholas Potteiger, Xenofon Koutsoukos
Department of Computer Science
Vanderbilt University
Nashville, TN, USA
{nicholas.potteiger, xenofon.koutsoukos}@vanderbilt.edu

2023 IEEE International Conference on Assured Autonomy (ICAA) | 979-8-3503-2601-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICAA58325.2023.00014

***Abstract*—Machine learning and reinforcement learning are increasingly used to solve complex tasks in autonomous systems. However, autonomous agents represented by large neural networks are not transparent leading to their assurability and trustworthiness becoming critical challenges. Large models also result in a lack of interpretability which causes severe obstacles related to trust in autonomous agents and human-machine teaming. In this paper, we leverage the hierarchical structure of behavior trees and hierarchical reinforcement learning to develop a neurosymbolic model architecture for autonomous agents. The proposed model, referred to as Evolving Behavior Trees (EBTs), integrates the required components to represent the learning tasks as well as the switching between tasks to achieve complex long-term goals. We design an agent for autonomous navigation and we evaluate the approach against a state-of-the-art hierarchical reinforcement learning method using a Maze Simulation Environment. The results show autonomous agents represented by EBTs can be trained efficiently. The approach incorporates explicit safety constraints into the model and incurs significantly fewer safety violations during training and execution. Further, the model provides explanations for the behavior of the autonomous agent by associating the state of the executing EBT with agent actions.**

***Index Terms*—explainable AI, behavior trees, hierarchical reinforcement learning, autonomous navigation**

I. Introduction

Autonomous systems are becoming more effective by using learning-enabled components (LECs) trained using machine learning (ML) and reinforcement learning (RL) methods. Autonomous agents used in complex safety-critical applications, such as autonomous driving, cyber-security, and healthcare, may cause unintended and harmful behavior if they are not carefully designed and used [1]. For such applications, agents must be trustworthy and explainable so that they can be incorporated into autonomous systems.

RL has been used extensively to perform autonomous tasks while maintaining safety [2]–[4]. RL agents typically use neural network models as function approximators trained using data-driven optimization methods to solve specific tasks. Existing algorithms work very well for short-term tasks with well-defined rewards. However, as tasks become longer and more complex, agents need to perform multiple subtasks that need to be coordinated at varying time scales. Learning models rely on an increasing number of parameters, are harder to train, and may even fail to to achieve long-term objectives with multiple tasks [5]–[7]. Large neural networks are not transparent leading to their assurability and trustworthiness becoming critical challenges. Large models also result in a lack of interpretability which causes severe obstacles related to trust in autonomous agents and human-machine teaming.

Autonomous agents need to complete multiple tasks over variable length time intervals. An agent performing a long-term task in a complex environment can decompose the task into multiple subtasks and determine how to coordinate these subtasks using the information received from the environment. This coordination can be achieved by creating a hierarchical organization of subtasks together with logical conditions that determine their execution. Autonomous agents designed to solve complex long-term goals can be represented by Behavior Trees (BTs) [8]. BTs have become a important tool in the design of autonomous agents in game AI and robotics, and are widely appreciated for their modularity and reactivity [9]–[11]. They provide an explicit hierarchical representation of the task and control structure of an agent based on components that are interpretable, modular, and reactive. Further, BTs can use LECs for representing the behaviors of a set of designated nodes and/or subtrees [12], [13]. Existing methods can be used for synthesizing the structure of BTs [12] and learning the behaviors of nodes [14].

Learning the parameters of the LECs can be accomplished using RL. In previous work, RL was shown to add flexibility and discover novel solutions to subtasks which is important when modeling an autonomous agent for a video game or robotics task. The approach presented in [11] uses a MAXQ approach to produce adaptive behaviors with constraints. Similarly, [10] presents an approach that learns behaviors using Q-Learning and incorporates them into a BT structure. Different methods and examples of BTs in robotics with AI algorithms are discussed in [8]. BT nodes realized as LECs have been used as execution and control behaviors for the popular video game *Minecraft* in [9]. Applications for traffic control flow of autonomous guided vehicles were explored in [15]. BTs with LECs are developed to play a minigame in *StarCraft 2* [16]. Hierarchical RL (HRL) is used to train components in BTs in [17]. A finite set of option tasks are predefined by the user, where the user specifies the purpose of each option using a task-specific reward function.

In this paper, we leverage the hierarchical structure of BTs and HRL to develop a neurosymbolic model architecture of

979-8-3503-2601-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICAA58325.2023.00014

an autonomous agent. The proposed model, referred to as Evolving BTs (EBTs), integrates LECs and standard components to represent the learning tasks of an autonomous agent as well as the switching between tasks that is required to achieve complex long-term goals. Our approach for training EBTs relies on goal-conditioned HRL, an RL method that decomposes the problem to well-defined components with multiple subgoals [5]–[7]. We consider autonomous navigation as a canonical problem for designing an agent using the proposed model. Autonomous navigation is a long-term problem for an agent that needs to reach an arbitrary goal position starting from any initial position by following waypoints or landmarks while maintaining safety by avoiding collisions with obstacles. The overall goal can be achieved by performing and coordinating short-term simpler subtasks.

The main technical contributions of our work are:

- We develop an approach for designing a safe explainable agent for autonomous navigation using EBTs trained using goal-conditioned HRL. The model incorporates multiple LECs that are integrated for achieving the long-term navigation goal while maintaining safety. In our approach, we design and optimize an explainable agent for autonomous navigation that performs interdependent learning tasks at different time scales that are jointly trained.
- We evaluate the approach against a state-of-the-art HRL method using a Maze Simulation Environment. The results show the autonomous agents represented by EBTs can be trained efficiently using HRL. The approach incorporates explicit safety constraints in the model and incurs significantly fewer safety violations both during training and execution. Further, the model provides explanations that HRL methods do not for the behavior of the autonomous agent by associating the state of the executing EBT with the actions determined by the LECs.

II. Problem Formulation

A. Autonomous Agent Design

Machine learning (ML) and reinforcement learning (RL) are increasingly used to solve complex tasks in autonomous systems. For example, hierarchical reinforcement learning (HRL) is used to design agents that perform autonomous navigation and robotic manipulation tasks. An agent learns how to make a sequence of decisions in an interactive environment through exploration via trial and error using feedback from its experiences. In RL, an agent aims to learn its policy in order to optimize the received rewards over a long-term perspective. Typical algorithms rely on function approximators such as deep neural networks that receive information from the environment and compute the optimal actions. Existing algorithms work very well for short-term tasks with well-defined rewards. However, complex long-term tasks require performing multiple subtasks that need to be coordinated at different time scales. Increasing the parameterization of neural network architectures for solving such complex problems results in fundamental limitations. The agents are not robust and cannot generalize to changes in the environment. In addition, large neural network models are not transparent leading to critical challenges arising in their assurability and trustworthiness. Large models also result in a lack of interpretability which causes severe obstacles related to trust in autonomous agents and human-machine teaming.

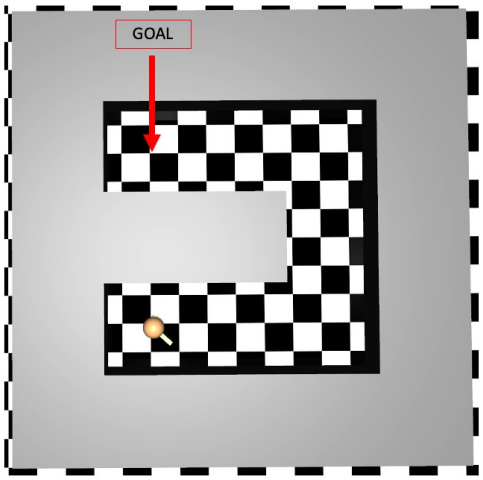


Fig. 1: Point Maze Autonomous Navigation Task

In general, autonomous agents need to complete multiple tasks over variable length time intervals. An agent performing a long-term task in a complex environment can decompose the task into multiple subtasks and determine how to coordinate these subtasks using the information received from the environment. This coordination can be achieved by creating a hierarchical organization of subtasks together with logical conditions that determine their execution. Well-defined subtasks can still be performed using learning-enabled components (LECs) trained, for example, using RL methods to cope with the uncertainty and variability of the real-world. Further, for safety critical applications, the agents can incorporate safety monitors and mitigation actions that are coordinated with the execution of the tasks.

In face of these challenges, the objective of the paper is to develop a neurosymbolic model representation for an autonomous agent that (1) employs LECs for performing specific subtasks, (2) enables switching between subtasks reacting to information received by the environment, and (3) captures an explicit hierarchy by incorporating the control flow required to execute the subtasks at varying time-scales. The model must integrate learned representations for the LECs and symbolic representations for coordination. Also, the model must allow for optimization through training. Specifically, in this paper we consider hierarchical RL methods for training. Further, after training, the model must be used for execution and evaluation of the autonomous agent.

B. Autonomous Navigation

We consider autonomous navigation as a canonical problem for designing an agent using the proposed neurosymbolic model. Autonomous navigation is viewed as a long-term problem for an agent that needs to reach an arbitrary goal position starting from any initial position by following waypoints or landmarks while maintaining safety by avoiding collisions with obstacles. In a safety critical application, such as autonomous driving, it is necessary that during execution the system maintains safety properties and models its current high-level behaviors for trustworthiness and for manual intervention in the case that the system exhibits unsafe behavior. The overall goal can be achieved by performing and coordinating short-term simpler subtasks.

We consider a specific example with a fixed "U" shaped *Point Maze* where the long-term task is to navigate to a goal position using an autonomous *point* agent as shown in Figure 1. The agent must determine the landmarks to be followed to reach the goal as well as the low-level continuous control actions to reach the landmarks. We use HRL to learn both optimal landmarks and control actions. Also, the agent must learn to avoid colliding with the barrier obstacles on the exterior of the maze. In a realistic navigation scenario, collisions will cause damage to the system or other environment factors. In summary, the agent must learn to reach any goal position in the maze environment while avoiding safety violations from collisions with barriers.

III. Background

A. Behavior Trees

The proposed neurosymbolic model is based on behavior trees (BTs). A BT is a tree structure used to model an autonomous agent which captures the hierarchy of various subtasks as well as the logical conditions for switching between them during execution. BTs have been used in various applications such as video game non-playable characters (NPCs) and robotics tasks [8], [9], [16] and offer important advantages for representing autonomous agents that include interpretability, modularity, and reactivity [8].

Interpretability is inherent to the structure of a BT. Components correspond to specific subtasks that need to be executed. These components are integrated into control structures that are identifiable as a sequential or logical decision process. Capturing such processes is beneficial for representing the switching between specific subtasks that are performed by an agent.

Modularity allows for re-use of subtasks as well as addition of new tasks in the BT. Modular subtasks are also transferable across different tasks with shared components, allowing for simple construction of new BTs. For example, in the autonomous navigation problem, we can add new subtasks in the BT representation of an agent for safety monitoring and mitigation.

Each subtask in a BT represents a sub-tree in the structure. BT's reactively switch to subtasks at each step of execution depending on logical conditions that can be evaluated using information received from the agent or environment. Subtasks can be paused and other higher-priority subtasks can be executed.

BTs are constructed using various types of tree nodes [8]. At each step of execution, one or multiple nodes will be ticked. Each node that is ticked returns a status of either *Success*, *Running*, or *Failure*. Figure 2 illustrates the visual representation for the nodes that are used in the construction of a learning-enabled BT in our approach.

There are two main types of BT nodes. The first are *execution* nodes. They can only be used as the leaf nodes of the BT and have two following subtypes:

- **Action** - This is a user-defined node programmed to complete a subtask. During execution, this node will return a status of *Running*. After execution, if the node is successful in completing the subtask, it will return a status of *Success*, otherwise *Failure*.
- **Condition** - A condition node returns a status of *Success* or *Failure*, depending on the respective result of a logical condition.

The second type of nodes are *control* nodes that determine the control flow of the BT during execution. Each *control* node has n children that determine its current status. Children can be either *execution* or other *control* nodes. The *control* nodes are of the following types:

- **Sequence** ($\rightarrow$) - A sequence node executes its children from left to right. If all children return *Success*, then the sequence returns *Success*. Otherwise, once one child returns *Failure*, *Failure* is returned by the sequence and execution terminates.
- **Fallback** ($?$) - This node executes its children left to right until one child node returns *Success* causing the fallback to return *Success*. Otherwise, if all child nodes return *Failure*, then the fallback returns *Failure*.
- **Decorators** ($\diamond$) - Decorators transform the status returned from one child. An example decorator is an *Inverter* ($\sim$) which transforms a status of *Success* to *Failure* and vice-versa. Another decorator is *RepeatUntilNSuccesses* (*RepN*) which returns *Success* only when its child has returned *Success* N times.

Each control node, except decorators, can incorporate *memory*. *Memory* maintains a history of which children have returned *Success*, so the control node does not need to continue status checking. Nodes without *memory* are useful for conditions that always need to be evaluated or reactive action nodes that require re-execution based on logical conditions.

B. BTs with Learning-enabled Components

LECs have been incorporated into execution and control nodes to represent behaviors that are learned [8]–[11]. The behavior of a node is represented by a LEC that produces actions dependent on information from the environment. A LEC action node is illustrated in Figure 2b. Although LECs could be trained using different ML methods, we consider

LECs trained using RL. Note also that BT nodes can share LECs between each other to increase efficiency in learning and complete long-term tasks using HRL [16], [17].

C. Goal-Conditioned Hierarchical Reinforcement Learning

We use HRL for training the neurosymbolic model of an agent. HRL has been shown to complete long-terms tasks in cases where standard RL algorithms fail [5]–[7]. Specifically, for training we use goal-conditioned HRL due to its explicit LECs for sub-tasks and explainable subgoals. Goal-conditioned HRL introduces a subgoal policy π_{sg} that assigns subgoals sampled from a goal-space to a low-level policy π_{low} to achieve. π_{sg} samples a subgoal every k timesteps, while π_{low} will generate an action every timestep. The intuition is that the low level policy learning to achieve short-term subgoals that ultimately lead to the long-term goal is easier than training to reach the actual goal.

A popular approach is HRL with off-policy correction (HIRO) [18], shown to improve sample efficiency and training success in complex long-term problems. HRL with a *k-step* adjacency constraint (HRAC) [19] expands on HIRO, by constraining sampled subgoals to be reachable in *k* actions from the current agent state [19]. This improves learning performance for long-term manipulation and navigation problems. HRL guided by landmarks (HIGL) [20] is an extension of HRAC which guides π_{sg} to produce explicit subgoals towards explored landmarks leading to a goal. During training, HIGL, building on previous work in state-space mapping using landmarks [21], samples landmarks based on coverage and novelty and constructs a graph from landmarks, the goal, and the agent's state. A regularization term is introduced for π_{sg} that encourages k-step adjacent subgoals towards the landmark on the shortest path to the goal. This landmark loss is a combination of the adjacency term with landmarks, replacing the need for the HRAC adjacency term in the subgoal policy loss. Then during execution, the subgoal and low-level policy are used with landmarks excluded.

Our approach for training the proposed neurosymbolic model is based on HIGL since it includes LECs for landmarks, subgoals, and low-level actions. The LECs are incorporated in the BT structure, and in addition, we incorporate an LEC for ensuring safety. The neurosymbolic model based on the BT structure represents explicitly switching between the various learned behaviors during execution of the autonomous agent.

IV. Evolving Behavior Trees

Developing an autonomous agent that can perform a complex long-term task requires coordination of multiple standard and learning-enabled components. Components must be integrated into an architecture that provides the control flow required for training and executing the agent. In this paper, we use Evolving Behavior Trees (EBTs) as the model representation for the architecture of an autonomous agent. The primary benefits of using EBTs lie in explicitly representing learning tasks using BT nodes together with the control structure that determines switching between tasks at varying time scales. EBTs allow for decomposing a complex ML architecture into a hierarchy of LECs together with the conditions and control required for their coordination. The model provides an interpretable representation of the agent. In addition, tasks captured as LECs are reusable and are transferable between EBTs to support different goals and missions.

EBTs provide mechanisms for capturing reactive switching between tasks at different time-scales based on well-defined logical conditions. In the autonomous navigation example, the agent decides when to update the next landmark on the path to the goal based on a logical condition that checks if the agent is within a certain proximity of the current landmark. The EBT node responsible for generating and following the landmarks can be implemented as a LEC which operates at a slower rate than the node for determining the optimal low-level control actions for moving safely towards a landmark.

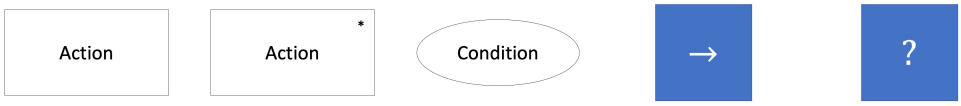
In the following, we describe the design process of EBTs for the autonomous navigation problem. We first focus on the learning tasks and associated LEC action nodes. Then, we construct an EBT for execution of the autonomous agent that utilizes the LECs. Finally, we explain how the LEC action nodes will be jointly optimized through a HRL training process.

A. Learning Tasks

We first define the long-term task for the autonomous agent. For the autonomous navigation problem, the task is to navigate between any initial position and final goal position while avoiding safety violations. The task can be accomplished using continuous control, a subgoal policy, and landmark generation as shown in [20]. These subtasks are performed using LECs that are represented as BT nodes. The subtasks need to be integrated into a control structure for generating optimal low-level control actions toward short-term subgoals in the direction of landmarks on the path to the final goal.

Low-level Control Policy LEC: The optimal low-level control is determined by the action node *SafeMoveTo*. The objective of the component is to determine the optimal action at every time step to move towards a short-term goal in the environment. The node can be implemented as a LEC that receives the current state of the agent and proximity sensor readings, and computes the optimal control action. Such a LEC can complete short-term tasks and can be used together with LECs that are responsible for generating subgoals and landmarks.

Subgoal Policy LEC: The objective of this component is to generate locations (referred to as subgoals) in the environment that can be reach in a fixed number of steps towards the target position. The component is implemented using a LEC as the action node *SafeGetSG* in the EBT. Specifically, *SafeGetSG* generates a subgoal towards the target position that can be reached in k-steps from the previous subgoal and not in close proximity to obstacles. *SafeGetSG* provides the subgoal to the low-level control policy LEC *SafeMoveTo* which runs for k times steps to move the agent towards the subgoal.



(a) Action (b) LEC Action (*) (c) Condition (d) Sequence (e) Fallback

Fig. 2: Base behavior tree nodes used in the formulation of our LEC action nodes and control structure in the Autonomous Agent EBT.

Landmark Generation LEC: To generate high-level landmarks for subgoals to move towards, we implement the LEC action node *CompLkGraph*. This node uses learned locations previously explored in the environment to construct a graph of landmarks. Using this graph, we can execute graph planning algorithms through landmarks to the goal. The landmarks are generated based on HIGL [20] which considers both coverage and novelty. Landmarks are spread around the environment. Novel locations are dissimilar from locations already seen in exploration, encouraging an agent, during training, to explore new areas.

Landmark Selection LEC: After generation of the graph, the objective is to select a landmark that is close to the agent and guides it to the goal. For this objective, we use an additional LEC action node, *GetLk*, to incorporate the agent's location and goal into the learned graph. Based on the shortest path to the goal, the node selects the landmark adjacent to the agent's location on the path. This process is repeated each time the agent reaches the current landmark until the agent reaches the goal. The selected landmarks are used to generate subgoals, which, in turn, are used to generate low-level actions.

B. Autonomous Agent EBT

The EBT governing the execution of the autonomous agent is shown in Figure 3 with a sequence node as the root of the tree. Initially, the landmark graph is computed once by executing *CompLkGraph*. Landmarks are generated only for coverage since exploration is not necessary as it is in training. After computing the graph, *CompLkGraph* is prevented from further execution using a logical condition represented by the condition node *LkGraph?* that will always return *Success* after execution of *CompLkGraph*.

After computing the landmark graph, the EBT switches to the next child that represents the traversal of the agent to the target position. Using a fallback node, the navigation process executes until the agent reaches the target position represented by the condition node *Done?*. A nested fallback either retrieves a new landmark or navigates towards the current one using subgoals from *SafeGetSG* and low-level actions from *SafeMoveTo*. A new landmark is retrieved using *GetLk* if the agent is within a certain proximity to the current landmark, represented by the *LkClose?*, returning *Success* if the logical condition is satisfied. Using a sequence, *GetLk* will only be executed if the previous child *LkClose?* returns *Success*.

C. Training

The learning tasks incorporated into the EBT prescribe the basic behaviors that need to be performed by the autonomous agent. The learning tasks can be realized by LECs that can handle the uncertainty and variability of the environment. The LECs are dependent on each other and they must be jointly trained.

The state s of the agent includes the agent's location g_{cur}, velocity, rotation, and sensor information. The low-level control policy $\pi_{low}(a|s, g; \theta_{low})$ used in *SafeMoveTo* determines the control action a towards a subgoal g_{sel} parameterized by θ_{low}. The reward function is based on the distance from the agent g_{cur} to the subgoal g_{sel} and includes a penalty C_{low} if the agent is close to a barrier and it is defined by

$$r^{low} = \begin{cases} -d(g_{cur}, g_{sel}) & \text{if } d(g_{cur}, o_i) \geq \epsilon \\ -d(g_{cur}, g_{sel}) - C_{low} & \text{if } d(g_{cur}, o_i) < \epsilon \end{cases}$$

where $o_i \in \mathcal{O}, \forall i \in |\mathcal{O}|$, $\mathcal{O}$ represent the boundaries of the obstacles, and $\epsilon > 0$ is a threshold for how close to an obstacle an agent is allowed to be.

The subgoal policy $\pi_{sg}(g|s; \theta_{sg})$ used in *SafeGetSG* generates a subgoal g_{sel} towards a landmark using state s and model parameters θ_{sg}. The reward function for the subgoal policy is defined based on the sum of the distances to the goal over a k-step time interval. A penalty C_{sg} is induced if a subgoal is located close to an obstacle. The definition of the the reward function is given by

$$r^{sg} = \begin{cases} -r_{sg,sum} & \text{if } d(g_{sel}, o_i) \geq \epsilon \\ -r_{sg,sum} - C_{sg} & \text{if } d(g_{sel}, o_i) < \epsilon \end{cases},$$

where

$$r_{sg,sum} = \sum_{i=1}^{k} d(g_{cur}, g_{final}). \quad (1)$$

Furthermore, the subgoal policy is optimized using a learning-enabled adjacency component to produce subgoals that can be reached in a fixed number of steps.

The subgoal policy is dependent on the generation of learned landmarks. *CompLkGraph* generates the landmarks based on the locations encountered from the low-level policy. As the agent explores the environment, the landmarks

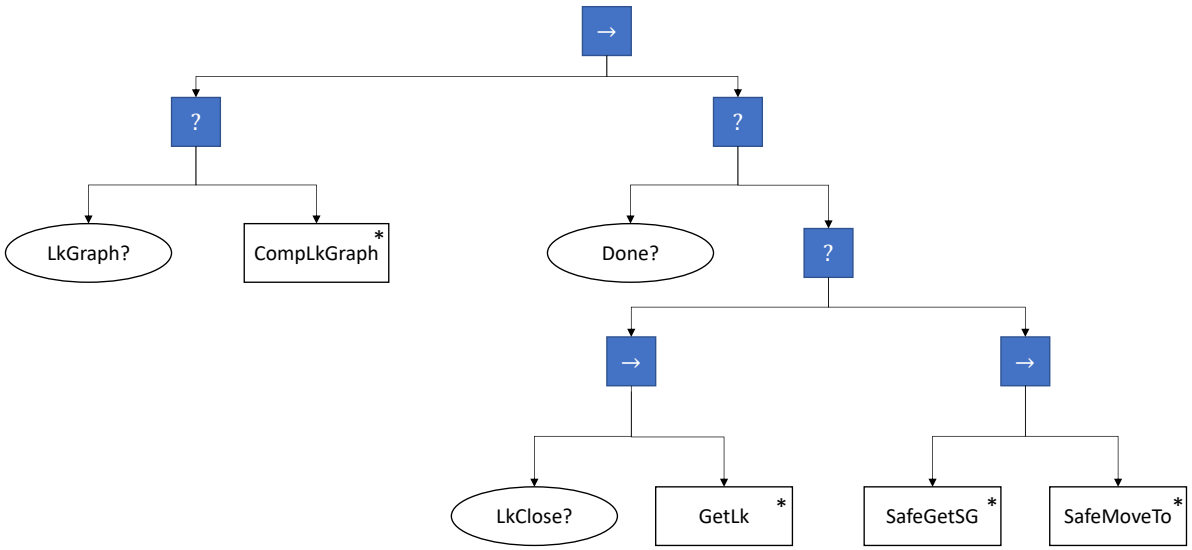


Fig. 3: EBT Model of the Autonomous Agent.

computed increasingly cover the reachable space. The coverage graph is determined using farthest point sampling [22]. Also, using a learning-enabled novelty component based on Random Network Distillation [23], novel locations, dissimilar from other locations, are stored as landmarks. The novelty of locations evolves over training, encouraging the subgoal policy to direct towards unexplored areas in the environment, possibly leading to new strategies to the goal. The collection of landmarks for coverage and novelty are then captured into a weighted graph where weights are distances between landmarks approximated using a value iteration technique utilizing the low-level policy [21]. Only landmarks that are within a certain distance have weighted edges, since the error increases as distance increases [21].

The landmark selection in *GetLk* is inherently related to the generation of the landmark graph in *CompLkGraph*. Hence, *GetLk* will also be dependent on the locations explored by the low-level policy. The node adds the agent's location g_{cur} and final goal g_{final} to the landmark graph using the same distance approximation technique. The shortest path is constructed from the learned landmark graph and a landmark is selected based on learned distance approximations.

V. Evaluation

We evaluate the training of the autonomous agent focusing on the efficiency of the training process with an additional training EBT. We also evaluate the performance of the agent EBT we constructed for execution in Figure 3. We compare with the state-of-the-art HRL approach HIGL [20]. We also extend HIGL by modifying the policies to incorporate safety

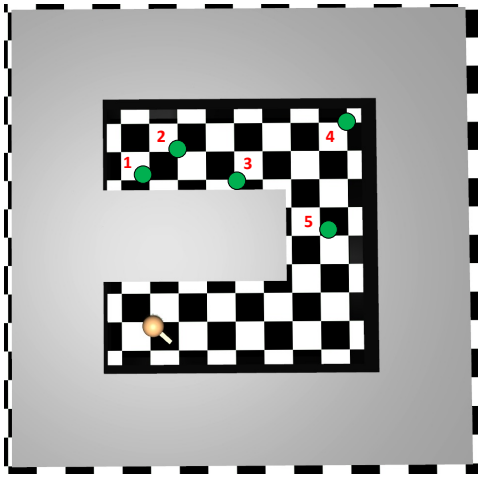


Fig. 4: Maze goals in execution of the agent chosen based on proximity to barriers and long-term objectives. The goals map to coordinates $[[-1, 7], [0, 8], [3, 6.5], [9.5, 9.5], [8, 4]]$ respectively. Goal 2 at $[0, 8]$ is the evaluation objective during training.

similarly to the LECs used in the EBT representation of the agent (HIGL Safe).

The difference between HIGL Safe and EBT is the explicit representation of the action and control flow using the EBT structure and the ability to extract high-level explanations from

the execution trace of an EBT. Therefore, we expect HIGL Safe and the EBT to exhibit similar quantitative performance with EBT representing the high-level behavior switching through its explainable execution.

A. Software and Hardware

We use the *PyTrees* framework to implement EBT structures, PyTorch [24] for optimization of LECs, and OpenAI Gym [25] for interaction with the *Point Maze* environment. The *Point Maze* environment is implemented in the MuJoCo Simulator [26]. The experiments are processed using a single GPU (RTX 3070) and 16 CPUs (11th Gen Intel Core i9-11900F, 2.50 GHz). *PyTrees*: https://github.com/splintered-reality/py_trees

B. Training Setup

For training, we develop an additional EBT that models the training process in HIGL Safe to compare training with BT nodes to standard approaches. We use the TD3 RL algorithm [27] with multi-layer perceptrons for both the low-level control and subgoal policies. To determine unsafe states for the agent, we utilize 8-dimensional proximity sensor readings. We choose $\epsilon = 0.02$ as our threshold for safety violations and $C_{low} = 10$ as the penalty during training.

For the subgoal policy, we choose a fixed amount of $k = 10$ time steps between sampling subgoals. To determine unsafe subgoals, we assume that we can detect if a subgoal is close to an obstacle. Similar to the low-level policy, the threshold and the penalty are set to $\epsilon = 0.03$ and $C_{sg} = 10$ respectively. The landmarks during training are retrieved from landmark graphs with 40 landmarks: 20 for coverage and 20 for novelty.

We use 1000 episodes for training the agent. Each episode is a simulation of the navigation task over 500 time steps, where a time step is incremented every tick *SafeMoveTo* executes a low-level action. The agent begins at location $[0, 0]$ and has to reach a random position $g_{final} \in [[-2, 10] \times [-2, 10]]$ using landmarks sampled at a safe distance $\delta = 0.5$ from the barriers. Evaluation towards a fixed goal at $[0, 8]$ is conducted every 5000 timesteps.

C. Training Performance

The results of the training are shown in Figure 5. The EBT and HIGL Safe achieve a higher level of success (100%) in fewer training time steps than the baseline HIGL as shown in Figure 5a. This is due to HIGL exploring unnecessary and unsafe routes that do not lead to the target position. For example, HIGL, at the beginning of training, can get stuck colliding against barriers. When the policies consider safety, the agent collides with fewer barriers and explores the space more efficiently, ultimately reaching the goal. EBT and HIGL Safe have similar performance.

Figure 5b shows that the EBT and HIGL Safe achieve lower safety violations over training. Furthermore, the decrease in safety violations is related to an increase in training success. In the EBT approach, once the agent learns to minimizes safety violations, it can explore the environment more efficiently. In HIGL, the agent does not learn to avoid barriers, encountering more safety violations, but after the agent learns to navigate to the goal, safety violations drastically decrease.

D. Execution Setup

To evaluate the execution of the autonomous agent, we use the optimized *ComputeLkGraph*, *GetLk*, *SafeGetSG*, and *SafeMoveTo* in the EBT in Figure 3. The landmark graph computed from *ComputeLkGraph* contains 20 landmarks based on coverage. Since, the agent is not in the exploration phase during training, we choose to not use the 20 novel landmarks. We evaluate the execution EBT using variable final goal positions. We choose five different positions in the environment given by $[[-1, 7], [0, 8], [3, 6.5], [9.5, 9.5], [8, 4]]$ as illustrated in Figure 4. For each goal and approach, we record the number of safety violations and time steps to reach the goal for 500 episodes.

E. Execution Performance

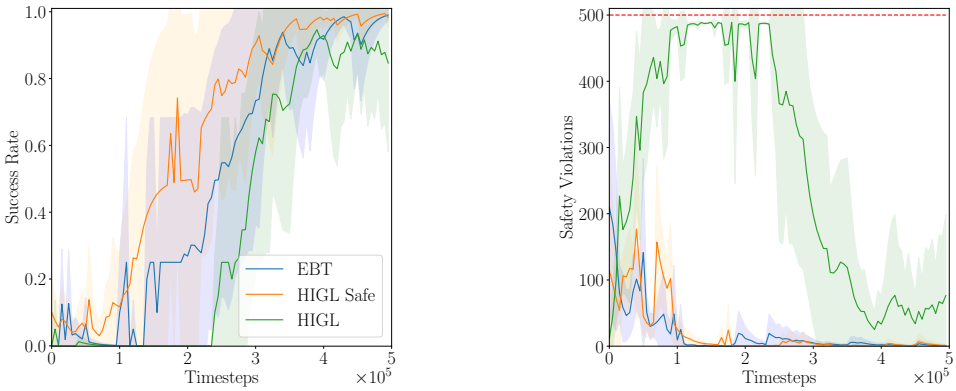
Our results in Table I demonstrate that both HIGL Safe and EBT achieve significantly lower safety violations per episode than HIGL. Using a two-sample t-test, we were able to determine that there is a statistically significant difference $(p < 0.001)$ between the mean safety violations of HIGL and both HIGL Safe and the EBT approach.

It is also interesting to note there is a statistically significant difference $(p < 0.001)$ between HIGL Safe and EBT for safety violations. The EBT consistently incurs less than 0.1 safety violations on average. In the case of HIGL Safe, when the goal is $[3, 6.5]$ the mean safety violations is 0.31, which is around one safety violation every three timesteps. This is a scenario where the agent needs to take tight turns. For the other goals, the EBT and HIGL Safe have a similar amount of safety violations incurred. Therefore, in most cases, these two approaches have similar performance. Since, the TD3 algorithm with the reward penalty in is used to optimize the policies in HIGL Safe and the EBT, we suspect that this anomalous scenario is due to stochasticity in the training runs. In either case, we will conduct further analysis into this scenario and conduct future work towards an optimization algorithm that guarantees safety during training and execution.

There is a trade off between safety and efficiency. The results show that HIGL is able to outperform HIGL Safe and the EBT with a difference in means of 20 timesteps. However, this is due to HIGL using unsafe routes that incur safety violations to reach the goal faster. The EBT and HIGL Safe are more conservative, avoiding safety violations by maintaining distance from barriers, leading to an increase in timesteps to reach the goal.

F. Explainability

The EBT in Figure 3 represents the architecture of the autonomous agent. During execution, the low-level actions can be explained based on the switching and status of the subtasks. In our implementation, we use the *PyTrees* framework to view the structure and execution status of the EBT at every



(a) Success Rate

(b) Safety Violations

Fig. 5: Evaluation over training 500k timesteps (1000 episodes) for the baseline HIGL, HIGL Safe, and EBT. Each datapoint represents (a) the rate the agent was successful in navigating to a goal at $[0, 8]$ or (b) the average number of safety violations over five evaluation episodes executed every 5000 timesteps. Since, each episode is 500 timesteps, there is a max of 500 safety violations per episode shown as a dashed red line (b). The solid lines and shaded regions represent mean and standard deviation over four training runs. For increased visual clarity we apply equal smoothing to all of the curves.

Agent Execution	**Safety Violations**			**Timesteps to Goal**		
Model	**Mean**	**Median**	**Std Dev**	**Mean**	**Median**	**Std Dev**
HIGL	8.1	6	7.45	**90.6**	**95**	34.7
HIGL Safe	0.06	0	0.4	112.3	111	33.2
EBT	**0.01**	**0**	**0.17**	111	113	**31.2**

TABLE I: Execution of the agent over 2500 episodes (500 for each of the five goals) for number of safety violations (collisions with barriers at each timestep) and timesteps to navigate to the goal.

tick. At a particular tick, the agent illustrates the state of the EBT with nodes that have succeeded, failed, running, or not currently executing. Associating the state of the executing EBT with the actions determined by the agent provides very useful explanations for the behavior of the autonomous agent especially compared with HIGL and HIGL Safe.

HIGL and HIGL Safe output a sequence of low-level control actions and subgoals over one episode to represent the execution process to reach the goal. In contrast, using the EBT representation, we obtain a sequence of EBT states paired with actions. The EBT states are explanations for the actions and the behavior or subtask associated with it. For example, in the computation of the landmarks, there is a trajectory of subgoals and low-level actions until the agent is close to the current landmark. We can associate this trajectory with the current landmark as the agent's objective is navigating to this landmark. The EBT states also represent the switching between the nodes *SafeMoveTo* and *SafeGetSG*, and reacting to the logical condition in *LkClose?* causing *GetLk* to execute producing a new landmark. In summary, by monitoring the EBT execution, low-level actions can be associated with subtasks performed by well-defined LECs providing providing explanations about the behavior of the autonomous agent.

Towards future work, a comparative study can be conducted between the BT approach and other approaches with a similar representation such as Hierarchical State Machines. To quantitatively evaluate the complexity of the models, we will look at using graph metrics, such as number of nodes, depth, or clusters. Further, one could conduct a qualitative study where participants describe and rate the explanation from the approach. These studies poses new challenges and as so we leave these for future research.

VI. Conclusion

The paper presents an EBT model for representing an autonomous agent trained with goal-conditioned HRL. The model incorporates multiple LECs that are integrated for achieving the long-term navigation goal while maintaining safety. The approach is evaluated using a maze simulation environment and the results show the autonomous agent represented by EBTs can be trained efficiently using HRL. Further, the model provides explanations for the behavior of the autonomous agent by associating the state of the executing EBT with the actions determined by the LECs.

A limitation of our approach is the manual design of the EBT structure which requires domain knowledge. In the

future, we will address this limitation by looking into methods, such as genetic programming, that can learn the optimal structure of the EBT. Another limitation is our approach minimizes safety violations during execution, but does not guarantee zero safety violations. Further work and analysis will need to be conducted to develop a safety component with theoretical guarantees. Also, we will extend our approach to realistic safety critical applications using underwater and aerial vehicles. Furthermore, we will investigate methods for analyzing the assurability, explainability, and trustworthiness of autonomous agents represented by EBTs.

REFERENCES

[1] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," 2016. [Online]. Available: https://arxiv.org/abs/1606.06565

[2] G. Thomas, Y. Luo, and T. Ma, "Safe reinforcement learning by imagining the near future," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 13 859–13 869.

[3] B. Thananjeyan, A. Balakrishna, S. Nair, M. Luo, K. Srinivasan, M. Hwang, J. E. Gonzalez, J. Ibarz, C. Finn, and K. Goldberg, "Recovery RL: safe reinforcement learning with learned recovery zones," *CoRR*, vol. abs/2010.15920, 2020. [Online]. Available: https://arxiv.org/abs/2010.15920

[4] B. Eysenbach, S. Gu, J. Ibarz, and S. Levine, "Leave no trace: Learning to reset for safe and autonomous reinforcement learning," 11 2017.

[5] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 1479–1487.

[6] O. Nachum, H. Tang, X. Lu, S. Gu, H. Lee, and S. Levine, "Why does hierarchy (sometimes) work so well in reinforcement learning?" *CoRR*, vol. abs/1909.10618, 2019. [Online]. Available: http://arxiv.org/abs/1909.10618

[7] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, p. 2778–2787.

[8] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI*. CRC Press, jul 2018. [Online]. Available: https://doi.org/10.12012F9780429489105

[9] M. Kartasev, "Integrating reinforcement learning into behavior trees by hierarchical composition," Master's thesis, 2019.

[10] R. Dey and C. Child, "Ql-bt: Enhancing behaviour tree design and implementation with q-learning," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, pp. 1–8.

[11] Q. Zhang, L. Sun, P. Jiao, and Q. Yin, "Combining behavior trees with maxq learning to facilitate cgfs behavior modeling," in *2017 4th International Conference on Systems and Informatics (ICSAI)*, 2017, pp. 525–531.

[12] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, 2018.

[13] D. Stojcsics, D. Boursinos, N. Mahadevan, X. Koutsoukos, and G. Karsai, "Fault-adaptive autonomy in systems with learning-enabled components," *Sensors*, vol. 21, no. 18, p. 6089, 2021.

[14] C. I. Sprague and P. Ögren, "Adding neural network controllers to behavior trees without destroying performance guarantees," *arXiv preprint arXiv:1809.10283*, 2018.

[15] H. Hu, X. Jia, K. Liu, and B. Sun, "Self-adaptive traffic control model with behavior trees and reinforcement learning for agv in industry 4.0," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 12, pp. 7968–7979, 2021.

[16] F. Lundberg, "Evaluating behavior tree integration in the option critic framework in starcraft 2 mini-games with training restricted by consumer-level hardware," Master's thesis, 2022.

[17] L. Li, L. Wang, Y. Li, and J. Sheng, "Mixed deep reinforcement learning-behavior tree for intelligent agents design," in *International Conference on Agents and Artificial Intelligence*, 2021.

[18] O. Nachum, S. S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," *Advances in neural information processing systems*, vol. 31, 2018.

[19] T. Zhang, S. Guo, T. Tan, X. Hu, and F. Chen, "Generating adjacency-constrained subgoals in hierarchical reinforcement learning," in *NeurIPS*, 2020.

[20] J. Kim, Y. Seo, and J. Shin, "Landmark-guided subgoal generation in hierarchical reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=IWhFd34QSSj

[21] Z. Huang, F. Liu, and H. Su, "Mapping state space using landmarks for universal goal reaching," in *Advances in Neural Information Processing Systems*, 2019, pp. 1940–1950.

[22] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[23] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1lJJnR5Ym

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[26] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.

[27] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 02 2018.