Improving the Performance of Backward Chained Behavior Trees that use Reinforcement Learning

Mart Kartášev, Justin Salér, and Petter Ögren

Abstract—In this paper we show how to improve the performance of backward chained behavior trees (BTs) that include policies trained with reinforcement learning (RL). BTs represent a hierarchical and modular way of combining control policies into higher level control policies. Backward chaining is a design principle for the construction of BTs that combines reactivity with goal directed actions in a structured way. The backward chained structure has also enabled convergence proofs for BTs, identifying a set of local conditions to be satisfied for the convergence of all trajectories to a set of desired goal states.

The key idea of this paper is to improve performance of backward chained BTs by using the conditions identified in a theoretical convergence proof to configure the RL problems for individual controllers. Specifically, previous analysis identified so-called active constraint conditions (ACCs), that should not be violated in order to avoid having to return to work on previously achieved subgoals. We propose a way to set up the RL problems, such that they do not only achieve each immediate subgoal, but also avoid violating the identified ACCs. The resulting performance improvement depends on how often ACC violations occurred before the change, and how much effort, in terms of execution time, was needed to re-achieve them. The proposed approach is illustrated in a dynamic simulation environment.

Index Terms—Behavior trees, Reinforcement learning, Autonomous systems, Artificial intelligence

I. INTRODUCTION

Behavior Trees (BTs) are a modular and hierarchical tool for combining a set of control policies into more complex high level control policies. Reinforcement learning (RL) on the other hand, is a method for creating control policies based on reward. Many problems have been solving using only RL, with so-called end-to-end learning [1]. Such an approach often requires a massive amount of computations and simulation time in more complex domains. However, very efficient algorithms already exist for some tasks, such as navigation from A to B, while others, such as grasping, are still open research problems. Therefore, it is reasonable to believe that in the coming years, many complex robotics problems will still need to be solved by a combination of manual and trained policies tailored for different sub-tasks. A BT is a way to combine such policies. It is therefore important to investigate how BTs could be combined with RL.

Robot control systems are often quite complex, and modularity is a well known tool for handling complexity. Ideally, different modules can be developed and tested separately and then put together to form a bigger system. Hierarchical modularity, where a module is made up of submodules, brings additional benefits as opposed to having a single layer of modules. This helps prevent very large modules or a very large unstructured set of modules, which often occur in large systems. BTs are a hierarchically modular tool for composing control policies. In fact, BTs have been shown to have minimal essential complexity [2] and could be thought of as an optimally modular decision structure. Modularity was also one of the primary reasons for the initial development of BTs in the computer game industry [3].

Fig. 1: A backward chained BT, the result of recursively replacing conditions with small BTs achieving those conditions, as illustrated in Figure 3. The ACC concept is illustrated for the action Chase Cow (yellow double rectangles). The convergence proof of [4] states that all actions need to achieve their postconditions, while not violating their so-called ACCs. Thus, when using RL to improve this action, there should be a positive award for achieving the postcondition Is close to cow (red double ovals) but also a negative reward for violating the ACCs, Safe from fire, safe from hostiles and has sword (green ovals).
In this paper we propose to use information about the structure of backward chained BTs, see Figure 1, when training controllers that are to be used in such BTs. Ideally, there are few dependencies between different parts of a system, but if a particular submodule reverses the achievements of previous submodules, the overall system is not likely to work well. Thus there is a need to identify the dependencies between different modules and create design specifications out of these dependencies to improve performance. We make use of previously published theoretical convergence proofs [4] to find such dependencies and use them to set up RL problems with consideration for earlier achievements.

To be specific, consider the example in Figure 1. The overall goal of the agent is to be safe from fire, safe from hostiles and not hungry. 

To satisfy the final objective, it is currently chasing a cow. The local goal is to get close to the cow in order to kill it. However, the convergence analysis in [4] shows that there is also a set of non-local goals that needs to be preserved. This includes not stepping into fire, staying away from hostiles, and keeping the sword that is later needed to kill the cow. If any of these are violated, the agent needs to stop chasing the cow and make sure they are satisfied before starting the chase again. Similar non-local constraints can be identified for any node, in any backward chained BT.

The main contribution of this paper is an approach for training RL policies that are to be used as subpolicies in a backward chained BT, informed by the convergence proof in [4]. This is done by shaping the reward in the RL formulation using the conditions identified by the proof. Thus the learning process will not only take into account the natural local subgoals, but also other, non-local constraints that are vital for the progress and performance of the overall BT policy, of which each RL policy is only a small part.

The outline of this paper is as follows. In Section II we describe the related work, and in Section III we give a background on backward chained BTs. Then the proposed approach is presented in Section IV, followed by experiments in Section V. Finally, conclusions can be found in Section VI.

II. RELATED WORK

In this section, we will describe earlier work investigating different aspects of combining learning with BTs [5]–[11]. There are a number of different ways of incorporating RL into a BT. One can either replace or design a single leaf node using RL, as was done in [5], or replace the interior composition nodes, sequence and fallback, with an RL node where the actions are the children of the original node, as was done in [6]–[8], [10]. One can also apply learning on the tree structure itself, moving, adding and deleting subtrees to improve the performance of the overall policy [12]–[15].

In [6] the authors aim to improve an existing BT. Given a reward function, the lowest level sequences of the BT are identified and used as actions in an RL problem. The q-value is estimated and used to create so-called q-conditions for each action, returning success when the corresponding action has the highest q-value. Using these, a BT is constructed that executes the proper action at the proper time.

In [5] two types of learning nodes are suggested, the learning action node and the learning composite node. In the learning action node, the user defines a complete RL problem, including states, actions and rewards. In the learning composite node, the actions are the children as either leaves or subtrees. The observation and the reward are defined by the user. In a similar spirit, [7] and [10], replace each fallback node with an RL-node, with user defined reward and the children as actions. The q-values are estimated, and the node executes the child with the highest q-value, given the current state. In [16], parameters throughout a BT, such as goal points of actions and threshold values of conditions, were learned simultaneously using an evolutionary strategy.

Strategies for improving the co-operative performance between sub-policies in a hierarchical policy have been investigated in [17] and [11]. In these works the focus is on improving the handover from one policy to the next in order to increase collaboration. While this can have the added effect of reducing certain types of failures, it does not directly avoid violating non-local constraints, which is the focus of this work.

The idea of learning the structure of a BT using genetic algorithms was first proposed in [12]. It was noted how the BT structure, with identical interfaces between subtrees on all levels, provided a policy representation that could easily be subjected to operations such as crossovers and mutations. These ideas were extended in [13] where grammatical evolution was used to enforce a given structure to the investigated BTs. Another approach for learning BT structures using genetic algorithms was used in [14]. There, the and-or-tree analogy of BTs was used to keep the size of the created trees down, while adding new functionality.

The approach proposed in this paper can be seen as an extension of the action node learning of [5], but goes beyond the work described above by looking at a particular subset of BTs with a backchained structure [18] and leveraging the convergence proof of such BTs in [4], to design individual learning problems that are adapted to the rest of the BT.

III. BACKGROUND

In this section we describe BTs, designing backchained BTs and results regarding their convergence.

A. Backward chained BTs

In this section we will describe the backward chained approach that was suggested in [18] and parts of the convergence analysis described in [4]. For a general description of BTs, we refer the reader to the survey [19] or the book [20].

Consider the example BT in Figure 2, designed to make sure that the agent has food. If the agent has food it will immediately return success, as the fallback (denoted by “?”)
only needs one child returning success to return success itself. If not, it will act to make the condition true. The subtrees for making it true are also children of the same fallback. Thus, if one option (Pick Apple) fails, or is not applicable, another option (Kill Cow) can be invoked. Both these actions have their own preconditions, describing when they can be invoked. These are collected in a sequence (denoted by “→”) as illustrated in Figure 2.

In general, given a list of the available actions, with preconditions and postconditions, such as the one in Table I, one can collect all actions having the same postconditions and create small BTs of the form shown in Figure 2.

**TABLE I: A list of available actions, with preconditions and postconditions.**

| Action       | Precondition          | Postcondition     |
|--------------|-----------------------|-------------------|
| Escape from fire |                        | Safe from fire    |
| Defeat hostile  |                        | Safe from hostiles|
| Eat           | Has food              | Not hungry        |
| Pick Apple    | Is close to apple     | Has food          |
| Kill Cow      | Is close to cow, has sword |                     |
| Craft sword   | Has materials, has crafting table |                     |
| Chase cow     | Can see cow           | Can see cow       |
| Search for cow|    ...                |                  |

The key idea of backward chained BTs is now to recursively apply the design in Figure 2, as illustrated in Figure 3. First, we list three top priority goals in a sequence node, safe from fire, safe from hostiles and not hungry. Then, instead of just checking the conditions, we can replace them (illustrated by dashed lines) with a small BT, of the form shown in Figure 2, to make them true. The resulting BT will have some new conditions, which in turn can be replaced by small BTs achieving them, and so on. This process is formally described in Algorithm 1.

**Algorithm 1: Creating a Backward Chained BT**

**Input:** Goals $C_1, \ldots, C_M$

1. $T_0 \leftarrow$ Sequence $(C_1, \ldots, C_M)$;
2. while Exists $C' \in T_0$ such that
   Parent($C'$) = Sequence AND $C'$ is postcondition of some action(s) $A_j$ do
   3. $T' \leftarrow$ Fallb($C'$, Seq(Pre($A_j$), $A_j$), ...
   4. Replace $C_i$ in $T_0$ with $T'$

The class of BTs we will analyse is defined as follows.

**Definition 1.** (Backward chained BT) A Backward chained BT is a BT that is constructed from a set of desired top level goal conditions in a Sequence, that are then recursively replaced by BTs achieving them, as described in Algorithm 1.

In [4], sufficient conditions for convergence of backchained BTs were presented. A key part of the requirements were that each action does not violate its own ACCs, defined as follows.

**Definition 2** (Active Constraint Conditions (ACC)). Given a BT $T_0$, and an action $A_i$ in that BT, the Active Constraint Conditions ACC(i) of $A_i$ are the set of conditions, apart from the preconditions of $A$, that need to return Success (be true) for $A$ to execute.

The ACCs of a specific action in a given tree can be found algorithmically. To determine ACCs, trace a path in the tree from the action in question to the root of the tree. For every sequence node along that path, select all the children to the left of the current path. Out of that selection take the condition nodes for those branches. The selected conditions correspond to the ACCs of the chosen action. For a more detailed description of the process, refer to [4].

Looking at Figure 1, we see that the ACCs of Chase Cow
have three ACCs: \textit{Safe from Fire}, \textit{Safe from Hostiles} and \textit{Has Sword}. Note that if one of these is violated, the agent has to switch to another action, to satisfy the condition again. If the agent comes too close to fire, it will invoke \textit{Escape fire}. If the agent is threatened by a hostile entity, it will invoke \textit{Defeat Hostile}, and if it breaks the sword it will have to make a new one. Thus, violating an ACC once might delay progress to achieving all goals, while repeated violation of ACCs might result in infinite loops where the agent is stuck undoing its actions. To illustrate the latter case we note that if \textit{Chase Cow} is completely unaware of fires, and there happens to be a fire between the agent and the cow, then the agent will move towards the cow, thereby getting too close to the fire, switch to \textit{Escape fire}, thereby moving away from the cow, switch back to \textit{Chase Cow} and again move towards the fire, and so on indefinitely.

Note that ACCs are not simply a list of all achieved subgoals. In order to craft the sword, the agent had to have materials and a crafting table. But once it has the sword, those subgoals are no longer important. Thus ACCs are the important subgoals that should not be violated.

The key idea of this paper is to use the information embedded in the ACC concept, taken from the proof in [4], in the definition of RL problems for individual actions inside a larger BT.

IV. PROPOSED APPROACH

In this section we will first state the problem we are trying to solve, and then provide the details of the solution we are proposing.

A. Problem

The problem we are addressing is what to do when a backchained BT created using Algorithm 1, that includes RL policies for some or all of the actions, does not succeed, i.e., it does not end up in a state where the overall BT returns success. Instead, it might end up in a state where the BT returns failure, or it keeps running indefinitely, switching amongst a set of actions.

Looking at the example in Figure 1 we could either have that \textit{Chase Cow} is never able to catch up with the cow, or that there are some hostile agents around and when chasing the cow the agent comes too close to the hostiles, resulting in a switch to \textit{Defeat hostile}, followed by a switch back to \textit{Chase Cow} and so on. As we will see below, this problem can be addressed in a structured way of making the agent chase the cow but at the same time try to stay away from hostiles.

B. Creating an action using RL

We propose to address the problem above by incorporating knowledge about the ACCs of the proof in [4] into the training of the individual nodes.

Thus, formulating the RL problem, we will not only use the desired postcondition (see Table I) as a positive reward, but also take ACCs (Definition 2) into account by applying a negative reward and/or ending the episode when an ACC is violated.

Definition 3 (RL problem for creating an action \( u_i \)). Let the reward be given by three constants \( M_p > 0 > M_t > M_{ACC} \), with \( p \) for postcondition and \( t \) for time, as

\[
R_a(x', x) = M_p \quad \text{if} \quad x' \in S_i
\]

\[
= M_{ACC} < 0 \quad \text{if} \quad x' \notin \bigcap_{j \in ACC(i)} S_j
\]

\[
= M_t < 0 \quad \text{if} \quad \text{else}
\]

The episode is ended when the postcondition is achieved, the agent dies, or the maximum number of time steps is reached. We will additionally evaluate the effect of ending the training episode when an ACC is violated.

Thus, a large reward \( M_p \) is given for achieving the objective, a negative reward \( M_{ACC} \) is given for violating the ACC, and a small negative reward \( M_t \) is given as time passes.

The reason for ending the episode upon ACC violation is as follows: During the normal execution of the BT, once the policy is trained, a violation of an ACC would by definition cause the BT to switch to a different action. Thereby it has the same effect as ending the episode - the action is forced to stop executing in favour of another action. Similarly, in this version of training, the episode is ended, but the environment is not reset. The training resumes with a new episode after the ACC violation has been resolved by the rest of the tree. This results in episodes that better represent the situations that the policy will encounter during real execution. Additionally, it stops the agent from accumulating further reward during the episode, making the impact of a single ACC violation larger.

C. When to apply ACC informed RL to actions

As stated above, the main problem we are addressing in this paper is improving the performance of an existing backward chained BT that has RL components. We concern ourselves with scenarios where the tree reaches a state where it returns failure, or executes for a very long time, looping through the same set of actions.

In light of the convergence proof of [4], one would then try to identify policies that violate their ACCs and make the training of those policies aware of the ACCs to train them accordingly to Definition 3.

Looking at Figure 1, we see that examples of ACC violations might be that \textit{Defeat hostile} keeps running into the fire and thereby initiates the execution of \textit{Escape fire}, or that \textit{Chase cow} moves too close to a hostile agent and thereby initiates the execution of \textit{Defeat hostile}.

Note that Definition 3 is different from the baseline approach of just running the RL based on the information in Table I, i.e., only trying to achieve the postconditions, without taking the ACCs into account. In the next section, we will compare the proposed approach with such a baseline method.

V. EXPERIMENTAL RESULTS

In this section, we will provide an empirical analysis of the method proposed above, identifying scenarios (such as Scenario 2) when it has significant impact on performance.
and scenarios (such as Scenario 1) when it has not. We first give an overview of the technical setup of the environment and experimental configurations and then present the results of our experiments. As will be seen, the benefits of taking ACCs into account vary with the context. If violating the earlier achieved objective can be remedied quickly (such as stepping away from the fire), and there is no risk of getting stuck in a pattern of repeatedly doing and undoing the same action, then the performance improvements will be very small. If however, it takes considerable time to achieve the objective (such as defeating a hostile agent), or there is risk of infinite loops (such as dropping the sword to catch the cow and then leaving the cow to get the sword), then the performance improvement can be significant. The experiments below will show examples of both these cases.

As described in Section II, the proposed approach can be seen as a variation of the action learning in [5], as it employs RL to a single action node, without changing the tree structure. Thus the principle described in [5] will be used for performance comparisons below. Note that it does not make sense to compare the approach to a standard backchaining setup [18] without RL as the outcome would heavily depend on the quality of the manual policies that the RL replaces. The same argument can be applied to any other high level planning approach, as they all depend on a set of existing low level skills/policies. Furthermore, a comparison to an end-to-end RL approach would just show how appropriate the given division into subpolicies is. Here, just as in [5], we assume that the division is fixed from the start, and motivated by external factors such as transparency, modularity, or the desire to reuse existing policies in parts of the tree. Finally, we are not aiming to compare different RL algorithms, but rather the effects of setting up RL problems in different ways. Therefore we solve all problems with the same algorithm, as described below. Thus, to investigate when the proposed approach is useful, we evaluate it in an existing backchained BT structure, and compare it to a standard RL solution to the same problem, as described in [5].

A. Technical setup

The technical setup for our experiments is based upon the simulation environment Project Malmö [21] and the RL algorithms in Stable-Baselines 3 [22]. These modules were chosen because they provide an extremely dynamic environment with countless objects and other agents that can be interacted with in numerous ways, while at the same time being freely accessible, enabling reproducible results. The source code of the resulting technical solution is available online at [23] alongside the results of the training.

Project Malmö [21] is an open-source platform developed by Microsoft Research, for the purpose of supporting AI experiments by providing a simulation environment based on the Minecraft engine. Malmö is a customisable virtual environment which allows for the testing of algorithms on tasks freely determined by the users. The Malmö interface allows for custom design of so called missions, which can be used to recreate common AI and Robotics related problem scenarios ranging from path-planning to multi-agent collaboration. One of the technical aspects of Malmö is that the control of the agent happens asynchronously from the stepping of the environment. This sets time constraints on the decision making process for an agent acting in the environment, in a way that is similar to a non-simulated system.

For RL we decided to use the open-source implementation Stable-Baselines 3 [22]. It is based on Open-AI Gym [24] and Baselines [25], and provides a customisable Gym interface which can be used to integrate it with many different environments, including Project Malmö. The central Gym interface allows the implementations to be integrated with different environments as well as easy switching between different RL algorithms, as long as they implement the Gym interface. At the time of writing, Stable-Baselines supports many widely known RL algorithms, such as PPO [26], DQN [27] and A2C [28]. We used PPO for the experiments of this paper.

B. Experimental setup

The primary aim of the experiments is to investigate to what extent does the setup in Definition 3 improve the efficiency of the BT execution. This will be measured in terms of the time needed to complete each mission.

We define our experiments based on the agent in Figure 1. Two different actions, Defeat hostile and Chase cow, will be trained using the RL problem in Definition 3, and the complete BT will be evaluated in two different scenarios, see Table II, one starting near a hostile, away from fire and not hungry, and the other starting hungry, but currently away from fire and hostiles.

TABLE II: The two scenarios to be evaluated, with remaining top level goal emphasized.

| Starting state                  |
|--------------------------------|
| Scenario 1 | Safe from fire, Not Safe from Hostiles, Not Hungry |
| Scenario 2 | Safe from fire, Safe from Hostiles, Hungry |

We will evaluate four different designs, see Table III. First one where the two actions are trained using a standard RL reward, taking only the desired postcondition into account. Then three different variations of RL that is aware of ACCs, following Definition 3. There are two ways to take the ACC into account. First using a negative reward \( M_{\text{ACC}} \), and second by ending the episode when the ACC is violated (this is what happens when the BT starts executing another action). We will evaluate all tree combinations of these two options, see Table III.

TABLE III: Parameters of RL problems

| Configuration                  | Reward for violating ACC \( M_{\text{ACC}} \) | End episode when violating ACC |
|-------------------------------|---------------------------------------------|---------------------------------|
| Standard RL [5]               | 0                                          | False                           |
| ACC Aware (Neg. Reward)       | -10                                         | False                           |
| ACC Aware (End Episode)       | 0                                           | True                            |
| ACC Aware (NR and EE)         | -1000                                       | True                            |

All configurations use \( M_p = 1000 \) and \( M_t = -0.1 \), see Definition 3.
As mentioned above, we train policies for two different BT actions, **Defeat hostile** and **Chase cow**. In Scenario 1, see Table II, the agent is not hungry, so **Chase cow** will not be executed, but in Scenario 2 both the trained actions might be used.

### TABLE IV: ACCs of the two actions to be trained.

| Action       | ACCs                           |
|--------------|--------------------------------|
| Defeat hostiles | Safe from fire                 |
| Chase cow     | Safe from fire, Safe from Hostiles, Has sword |

The ACCs of the two actions to be trained can be seen in Table IV. Thus, when defeating hostiles the agent should try to keep out of the fire, while when chasing the cow it should try to keep out of the fire, keep out of the way of hostiles, and avoid losing the sword.

### C. RL Configuration

The training for both of the RL policies, **Defeat hostile** and **Chase cow**, was performed with the *Stable baselines* [22] implementation of PPO [26]. Both the policy and the value function were modelled as a Multi-Layer Perceptron, with default parameters from Stable-Baselines used during the training process.

The architecture for both the value and policy networks is a fully connected 2-layer network of 64 neurons per layer. This is preceded by a observation input layer and succeeded by the action layer, depending on the action and observation spaces of each agent.

The action spaces are as follows (bold for **Chase cow** only):
- Move forward
- Move backward
- Turn left
- Turn right
- Pitch up
- Pitch down
- Stop (NO-OP)
- Attack

The observation spaces are feature vectors engineered from data available through the Malmö back-end, containing the following values (bold for **Chase cow** only, emphasis for **Defeat Hostile** only):
- Health (scalar)
- Satiation/Hunger (scalar)
- Surrounding terrain block grid (11x11x1 matrix)
- Enemy relative position (polar co-ordinates)
- Enemy health (scalar)
- Enemy targeted (boolean)
- Entity (cow) relative position (polar co-ordinates)
- Entity targeted (boolean)

### D. Training Results

Each RL policy was trained for 2 million timesteps, using the configurations described in Table III. The training results are shown in Figures 4a and 4b.

It can be seen that the training has converged to policies that consistently produce good rewards in all of the respective configurations. There is a larger variance of rewards in the later stages of training for the **Defeat hostile** policies in Figure 4a. This is caused by the fact that defeating hostiles is a very dynamic activity that can play out in many different ways. Even a small mistake can have the agent pushed into the fire by the opposing agent. As the training experiments with small random deviations from the currently known best policy in the later training stages, it can lead to relatively large deviations in accumulated reward during training, as small changes in behavior can quickly lead to large negative rewards. There is a smaller variance in **Chase cow**, as the hostile is mostly static while not having seen the agent.

### E. Evaluation Results

To see the effect on the overall performance of the BTs, including trained actions, we ran evaluations for 1000 episodes in the two scenarios described in Table II.

The numerical performance metric, listed in Tables V and VI, is the time taken to complete the mission. Additional information in terms of the number of ACC violations, both...
TABLE V: Evaluation of Scenario 1, starting not hungry but close to hostile. Note how there is no clear performance improvement, in terms of time to completion, between the different methods. There is an increase in episode failure rate when ACCs are not considered.

| Configuration                      | ACC* violations (% episodes) | ACC* violations (# steps) | Time (steps to completion) |
|------------------------------------|------------------------------|---------------------------|---------------------------|
|                                    | %                            | Mean | SD      | Mean | SD      |
| Standard RL [5]                    | 20.8%                        | 5.83%| 2.1    | 5.24 | 446    | 78    |
| ACC aware RL (Neg. Reward)         | 5.60%                        | 0.6% | 0.4    | 1.96 | 422    | 77    |
| ACC aware RL (End Episode)         | 3.40%                        | 0.6% | 0.24   | 1.39 | 437    | 85    |
| ACC aware RL (NR and EE)           | 2.90%                        | 0.2% | 0.15   | 0.94 | 460    | 76    |

*ACC = "Safe from Hostiles"

Looking at the data from Scenario 1 in Table V, we see a very clear performance improvement from using the ACC-aware RL configuration where ACCs are not considered, in Scenario 1. This corresponds to the critical nature of the "Safe from Fire" ACC. Since fire reduces health very rapidly in the game, it poses a significant risk to the agent. Therefore, violating the ACC quickly leads to mission failure. Even though we see that the percentage of ACC violations is generally not on the same level as the percentage of failed episodes in Table V, there is still a significant increase in the failure rate of the standard RL configuration. Since violating this ACC can quickly lead to failure, the agent would be expected to learn which states can lead to the negative reward associated with failure over time. However, it seems that accounting for the ACC explicitly in these failure critical scenarios might allow the agent to learn more efficiently. A more thorough numerical evaluation of this phenomenon would be necessary in order to determine and measure the persistence and magnitude of this effect, which is not within the scope of this paper.

Looking at the data from Scenario 2 in Table VI, we see that the differences in completion time are fairly large (more than 500%), with the ACC-aware (NR and EE) configuration performing best, followed by ACC-aware (End Episode) and ACC-aware (Neg. Reward). We can also see that for the standard RL [5] configuration the percentage of episodes with at least one ACC violation is very high (100%), and in each episode over 50% of the total number of time steps (790 time steps out of 952) violate ACCs. Thus, in Scenario 2 we see a very clear performance improvement from using the ACCs. The ACC violation of Safe from Hostiles leads to the hostile agent approaching the agent and a subsequent fight. Even though all agents are able to eventually defeat the hostile, the fight itself creates a significant delay in reaching the cow.

Looking at the difference between the three ACC-aware configurations in Tables V and VI we note the following. The negative reward alone has an effect on the ACC violation rate, but on average has a smaller impact than ending the episode. It could be argued that larger negative rewards might have a larger impact, but it could theoretically lead to instability if it is too large, as this reward would be given for every timestep in which the ACC is violated.

The "End Episode" configuration has a strong effect on ACC violations in both scenarios. This sort of ending to the episode is always a negative occurrence, because the potential positive reward can no longer be reached, leaving the agent with a lower cumulative reward. Using the configuration of "End Episode" and "Negative Reward" together appears to have the largest overall effect. This option not only prevents future progress towards a positive reward but also lowers the reward accumulated during the episode, without the risk of instability as the large reward is only given once.

To summarise, we note that avoiding ACC violations can have either a very small or a very large impact on the overall
performance of the BT, in a way that depends on how much extra effort, in terms of execution time, is needed to re-
achieve the ACC. Furthermore, the strongest effect in terms of reducing ACC violations is given by the combination of a negative reward and ending the episode.

VI. CONCLUSIONS

This paper investigated the use of ACC aware RL policies for improving the performance of backward chained BTs with RL actions. The key idea was to use the ACCs, which were defined in the literature as an important part of theoretical convergence proofs, in the setup of the RL problems for individual actions. Our experiments show that the proposed approach can produce significant performance improvements in cases where achieving the ACC takes significant effort in terms of execution time (such as defeating a hostile).
The improvements in terms of time can be smaller in cases where achieving the ACC requires less effort (stepping away from fire) or there is no risk of getting caught in a loop of repeatedly doing and undoing the same action.

Finally, we conclude that ACCs can be used to both analyse the BT, when considering the implementation of an RL based action and also to provide inputs to the reward function and the training process itself.

ACKNOWLEDGEMENT

This work was supported by Stiftelsen for Strategisk Forsknings (SSF) through the Swedish Maritime Robotics Center (SMaRC) (IRC15-0046).

REFERENCES

[1] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al., “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” Nature, vol. 575, no. 7782, pp. 350–354, 2019.

[2] O. Biggar, M. Zamani, and I. Shames, “On modularity in reactive control architectures, with an application to formal verification,” ACM Trans. Cyber-Phys. Syst., vol. 6, no. 2, apr 2022. [Online]. Available: https://doi.org/10.1145/3511606

[3] M. Mateas and A. Stern, “Façade: An experiment in building a fully-realized interactive drama,” in Game developers conference, vol. 2, 2003, pp. 4–8.

[4] P. Ögren, “Convergence analysis of hybrid control systems in the form of backward chained behavior trees,” IEEE Robotics and Automation Letters, vol. 5, no. 4, pp. 6073–6080, 2020.

[5] R. d. P. Pereira and P. M. Engel, “A framework for constrained and adaptive behavior-based agents,” arXiv preprint arXiv:1506.02312, 2015.

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

[7] Y. Fu, L. Qin, and Q. Yin, “A reinforcement learning behavior tree framework for game ai,” in Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering, 2016.

[8] B. Hannaford, D. Hu, D. Zhang, and Y. Li, “Simulation results on selector adaptation in behavior trees,” arXiv preprint arXiv:1606.09219, 2016.

[9] C. I. Sprague and P. Ögren, “Adding neural network controllers to behavior trees without destroying performance guarantees,” in 2022 IEEE 61st Conference on Decision and Control (CDC), 2022, pp. 3989–3996.

[10] 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). IEEE, 2017, pp. 525–531.

[11] M. Kartasev and P. Ögren, “Improving the performance of learned controllers in behavior trees using value function estimates at switching boundaries,” arXiv preprint arXiv:2305.18903, 2023.

[12] C.-U. Lim, R. Baumgarten, and S. Colton, “Evolving behaviour trees for the commercial game defcon,” in European conference on the applications of evolutionary computation. Springer, 2010, pp. 100–110.

[13] M. Nicolau, D. Perez-Liebana, M. O’Neill, and A. Brabazon, “Evolutionary behavior tree approaches for navigating platform games,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 9, no. 3, pp. 227–238, 2016.

[14] 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.

[15] C. Paduraru and M. Paduraru, “Automatic difficulty management and testing in games using a framework based on behavior trees and genetic algorithms,” in 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE, 2019, pp. 170–179.

[16] M. Mayr, K. Chatzilygeroudis, F. Ahmad, L. Nardi, and V. Krueger, “Learning of parameters in behavior trees for movement skills,” IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021.

[17] J. Erskine and C. Lehner, “Developing cooperative policies for multi-stage reinforcement learning tasks,” IEEE Robotics and Automation Letters, vol. 7, no. 3, pp. 6590–6597, 2022.

[18] M. Colledanchise, D. Almeida, and P. Ögren, “Towards blended reactive planning and acting using behavior trees,” in 2019 International Conference on Robotics and Automation (ICRA). IEEE, 2019, pp. 8839–8845.

[19] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, “A survey of behavior trees in robotics and ai,” Robotics and Autonomous Systems, vol. 154, p. 104096, 2022.

[20] M. Colledanchise and P. Ögren, Behavior trees in robotics and AI: An introduction., CRC Press, 2018.

[21] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, “The malmo platform for artificial intelligence experimentation,” in Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, ser. IJCAI’16. AAAI Press, 2016, p. 4246–4247.

[22] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, “Stable baselines3,” https://github.com/DLR-RM/stable-baselines3, 2019.

[23] M. Kartasev and S. J. Ramberg, “Repository: Improving the performance of backward chained behavior trees using reinforcement learning,” https://github.com/martkxnts/BTBackchainingRL, 2023.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” arXiv preprint arXiv:1606.01540, 2016.

[25] P. Dharirali, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, “Openai baselines,” https://github.com/openai/baselines, 2017.

[26] J. Schulman, J. Harb, and D. Abbeel, “Proximal policy optimization algorithms,” arXiv preprint arXiv:1707.06347, 2017.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ser. ICML’16. JMLR.org, 2016, p. 1928–1937.