Autonomous Acquisition of Behavior Trees for Robot Control

Bikramjit Banerjee
School of Computing
The University of Southern Mississippi
Hattiesburg, MS, 39406
Bikramjit.Banerjee@usm.edu

***Abstract*— Behavior trees (BT) are a popular control architecture in the computer game industry, and have been more recently applied in robotics. One open question is how can intelligent agents/robots autonomously acquire their behavior trees for task level control? In contrast with existing approaches that either refine an initially given BT, or directly build the BT based on human feedback/demonstration, we leverage reinforcement learning (RL) that allows robots to autonomously learn control policies by repeated task interaction, but often expressed in a language more difficult to interpret than BTs. The learned control policy is then converted to a behavior tree via our proposed *decanonicalization* algorithm. The feasibility of this idea is based on a proposed notion of *canonical behavior trees* (CBT). In particular, we show (1) CBTs are sufficiently expressive to capture RL control policies, and (2) that RL can be independent of an optimal behavior permutation, despite the BT convention of left-to-right priority, thus obviating the need for a combinatorial search. Two evaluation domains help illustrate our approach.**

I. INTRODUCTION

Many control architectures for autonomous robots have been studied over the decades. Finite state machines (FSM) and hierarchical finite state machine (HFSM) underlie the commonest architectures, because of their intuitive structures and modularity of HFSMs. Algorithms also exist for optimization and learning of such machines [1], [2]. However, these architectures tend to become unwieldy with increasing complexity, which prompted the computer game industry to seek alternatives for non-player characters more than a decade ago, leading to the adoption of behavior trees (BT) [3]. Another disadvantage of HFSMs is the fact that behaviors that are more abstract (e.g. approach ball) operate on a different time scale than more ground/primitive level behaviors (e.g., turn left by 2 degrees). On the one hand, this typically restricts planning to behaviors at the same level of abstraction. On the other hand, commitment to a higher level behavior throughout its duration reduces the robot's ability to react to exigencies—a problem solved by teleo-reactive programs [4] which were a precursor to BTs.

Behavior based architecture (BBA) [5], another popular architecture for robot control, derives inspiration from Brooks' subsumption architecture [6]. BBAs not only feature modularity, scalability and code reuse, but they also solve the time scale problem since all behaviors, irrespective of their level of abstraction, operate on the same time scale. BTs are a derivative of the BBA, and although they were initially popularized in the game industry, they are being increasingly adopted for robot control [7], [8], [9].

In this paper, we consider how an agent can autonomously acquire a BT for task level control. Unfortunately, a rather limited amount of work exists on this topic. Colledanchise et al. [10] apply evolutionary approaches to incrementally construct a BT, that requires an externally supplied fitness function for the entire tree (or it can be estimated by offline simulations, but this slows down learning). By contrast, we consider reinforcement learning (RL), which only requires a reward function constructed at the task level—often an easier proposition. Dey & Child [11] apply RL to refine an already constructed BT, whereas we do not require an initial BT. Pereira & Engel [12] incorporate RL into individual nodes (which can be viewed as skills), where users pick the RL parameters to allow the agent improve those skills. This, again, requires an initial BT be given. Robertson & Watson [13] describe an algorithm for incrementally constructing a BT from experience traces produced by an expert or a human guide. By contrast, we enable an agent to autonomously acquire its BT without requiring human/expert demonstration.

Instead of incrementally growing a tree or refining an externally supplied tree, we propose to directly convert an autonomously acquired RL control policy into a BT. The motivation behind this idea is the following. RL policies are often represented in ways difficult for human interpretation, e.g., using neural networks or other function approximators. By contrast, a BT is easier for human users to understand and modify, protecting human supervision in a robotic society. However, initial construction of a BT from scratch may be tedious to a human, which our proposal ameliorates. The proposal hinges on the notion of a canonical behavior tree (CBT). A CBT only contains (user coded) behaviors, and guards/tests, without any structural variety, making it trivial to construct for RL. Moreover, we prove theoretically that even the guards/tests can be constructed *after* RL, meaning that RL can be conducted without a priori knowledge of an optimal behavior ordering, despite the BT convention of left-to-right priority, thus obviating the need for a combinatorial search. We also prove that an optimal RL policy based on a given set of behaviors is indeed expressible as a CBT, thus establishing the feasibility of the proposed approach. We propose a *decanonicalization* algorithm to convert the learned CBT into a BT that is perhaps easier for a human

to understand, and something close to what a robot control designer might build by hand. We illustrate our approach with two versions of a package delivery domain in the Gazebo simulator, where an iRobot Create 2 loads pacakges from one location and unloads them at another, but taking a recharge detour when it senses low battery.

II. Background

A. Behavior Trees

A BT is a directed rooted tree, with internal nodes serving to direct control flow, while leaf nodes represent action execution or condition evaluation. The execution of a BT starts from the root node which generates clock pulses or *ticks* at a given frequency, which are then sent to its children. A parent passes on a tick to its children following logic that depends on its type. A child, which may pass on the tick to its children, must return to the parent one of three status: *running* (if its execution is on-going), or *success* (if its goal has been achieved) or *failure*. Internal, or control flow nodes, may be of three main types: *Selector*, *Sequence* or *Parallel*. A Selector node routes a tick to its children from left to right until it finds a child that returns either *success* or *running*, which is the status it then returns back to its parent. The remaining children do not receive the tick. A Selector node returns *failure* iff all of its children return *failure*. A Sequence node routes a tick to its children from left to right until it finds a child that returns either *failure* or *running*, which is the status it then returns back to its parent. The remaining children do not receive the tick. A Sequence node returns *success* iff all of its children return *success*. A Parallel node can cause the simultaneous execution of multiple children, which can be useful for robots with multiple actuators, but we defer this topic to future work and ignore parallel nodes in this paper [1]. The leaf nodes can be of two types: *Condition* and *Action*. When a Condition node receives a tick, it tests a proposition and returns *success* or *failure* depending on its evaluation. It never returns *running*. An Action node uses a tick to execute one step of some actuator command. It returns *success* if/when the (multi-step) action completes correctly, *failure* if it fails, or *running* if it needs more steps to complete. Actions are, in fact, behaviors. There is an additional type of control flow node, called the *Decorator*, that has a single child, and simply modifies the return value of its child according to some user defined logic (e.g., negation), before passing the modified value to its parent. Decorators improve human readability and the expressive power of BTs, but we defer them to future extension of this work in more complex domains. Figure 1 shows the various node types considered in this paper.

A BT allows the definition of hierarchical activities, as its internal nodes can be seen as composing lower level activities with AND-OR logic into higher level (more abstract) activities.

[1]Note that multi-actuator scenarios can also be addressed with separate BTs for each actuator type, e.g., wheels vs. arms, thus not requiring parallel nodes.

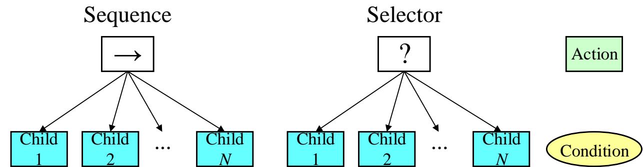


Fig. 1. Various node types of a behavior tree. Some node types (viz., Parallel and Decorator) are omitted.

B. Reinforcement Learning

Reinforcement learning problems are modeled as *Markov Decision Processes* or MDPs [14]. An MDP is given by the tuple $\langle \Sigma, A, R, P \rangle$, where Σ is the set of visible environmental states that an agent can be in at any given time, A is the set of actions it can choose from at any state, $R : \Sigma \times A \mapsto \Re$ is the reward function, i.e., $R(\sigma, a)$ specifies the reward from the environment that the agent gets for executing action $a \in A$ in state $\sigma \in \Sigma$; $P : \Sigma \times A \times \Sigma \mapsto [0, 1]$ is the state transition probability function specifying the probability of the next state in the Markov chain following the agent's selection of an action in a state. The agent's goal is to learn a policy $\pi : \Sigma \mapsto A$ that maximizes the sum of current and future rewards from any state σ, given by, $V^\pi(\sigma^0) = E_P[R(\sigma^0, \pi(\sigma^0)) + R(\sigma^1, \pi(\sigma^1)) + \ldots]$ where $\sigma^0, \sigma^1, \ldots$ are successive samplings from the distribution P following the Markov chain with policy π.

Reinforcement learning algorithms often evaluate an action-quality value function Q given by

$$Q(\sigma, a) = R(\sigma, a) + \max_\pi \gamma \sum_{\sigma'} P(\sigma, a, \sigma') V^\pi(\sigma') \qquad (1)$$

This quality value stands for the sum of rewards obtained when the agent starts from state σ at step t, executes action a, and follows the optimal policy thereafter. Model free methods directly learn $Q(\sigma, a)$, often by online dynamic programming, e.g., *Q-learning* [14]. The final policy is calculated as

$$\pi(\sigma) = \arg\max_a Q(\sigma, a). \qquad (2)$$

In this paper, we focus on tasks that allow $Q(\sigma, a)$ values to be stored in a table, i.e., tabular Q-learning. In such tasks, the state space is simple enough to not require any function approximation.

C. Options

Sutton et al. [15] extended the theory of RL in MDPs to *semi*-MDPs (SMDP) with temporally extended actions, called *options*. An option is given as a tuple $O = \langle I_O, \pi_O, \beta_O \rangle$, where $I_O \subseteq \Sigma$ is a set of states where the option can be initiated, $\pi_O : \Sigma \mapsto A$ is a local option policy mapping states to primitive actions (or other options), and $\beta_O : \Sigma \mapsto [0, 1]$ is a termination condition for the option. A primitive action, a, is a special case of options under the setting $I_a = \Sigma$, $\beta_a(.) = 1$ and $\pi_a(.) = a$. In this paper, we also consider actions that can be complex, multi-step activities, called *behaviors*. We shall later show the equivalence of behaviors and deterministic options.

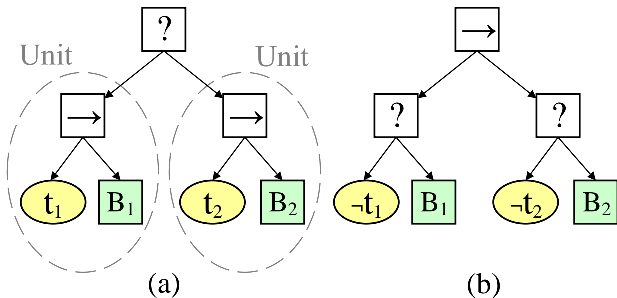


Fig. 2. Equivalent behavior trees.

In robotics domains, the state space Σ is often expressed as a product space over a set of real-valued features $\mathcal{F} = \{f_1, \ldots, f_k\}$. Each option/behavior policy π_i will typically map a subset of these features $\mathcal{F}_i \subseteq \mathcal{F}$ to atomic actions. We assume that a set of predicates over $\mathcal{F}$, $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_P\}$, is given for the construction of BT conditions. For instance, while the battery-level is a real valued feature, $battery_low(\sigma)$ is a predicate. Then the top-level policy $\pi : \prod_i \mathcal{P}_i \mapsto \mathcal{O}$ maps combinations of predicates over Σ to set of options, $\mathcal{O}$. It is this top-level policy that a robot can learn via RL and then convert to a BT.

Apart from the above description of options, we also refer to the definition of a behavior B as a tuple $B = \langle \pi_B, \rho_B, \Delta t \rangle$ [16], where

- π_B refers to the local control policy [2] of B that maps states to (a distribution over) atomic actions. This can be likened to the option policy π_O described above.
- ρ_B maps states to a return status $\in \{success, failure, running\}$.
- Δt stands for the time step, or tick duration, which may be the same for all behaviors.

III. Learning Behavior Trees

Our main objective is for an agent to autonomously acquire its behavior tree via ground interactions in the task domain. While the structure of a behavior tree is highly flexible to foster development by a human user, the flexibility creates a challenge in autonomous acquisition by an agent. In particular, there are multiple ways of representing essentially the same tree. Figure 2 shows an example. This can be a challenge to learning mechanisms that often work by incremental structural changes that improve some measure of performance. When different sequences of changes to a given behavior tree policy can lead to the same (or equivalent) policy, the repetitiveness can raise the complexity of a search in the space of policies.

Our proposal is to canonicalize the structure of behavior trees, to avoid or reduce the repetitiveness described above. Given that behavior trees are essentially AND-OR trees, and inspired by the universal normal form of logic which are depth-2 representations, we propose to reduce behavior trees to a depth-2 representation. The root is always a selector node where ticks originate. Its children are all sequence nodes. Each sequence node has two children. The left child is a condition node, and the right child is a behavior node. The condition can be thought of as the *guard* for the behavior, i.e., a test for all conditions such that the behavior will be executed iff the test returns true/success. Figure 2(a) shows a small example of this reduced representation, which we call the Canonicalized Behavior Tree (CBT). Each sequence node at depth-1 with its two children forms a *unit*. Thus a CBT can be seen as a selection among a number of units. The overall idea is that developers will only write the code for behaviors, and then let the agent autonomously acquire the guards assuming that the units are arranged in the CBT in an arbitrary order. To produce the final behavior tree that is human understandable and closer to something that a human user might create manually, the autonomously acquired guards within the units may need to be "de-canonicalized", which will possibly disperse the behaviors to different depths in the final tree.

[2] This is presented as a control law in [16], i.e., the right side of a difference equation, but we use a different representation to be consistent with RL literature.

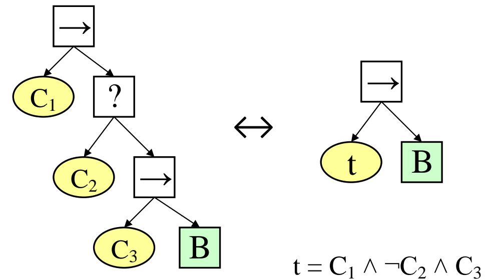


Fig. 3. A BT and its reduction to a unit.

There are several questions regarding the feasibility of the above broad idea, which we address in turn in the following sections.

1) Can *all* behavior trees be canonicalized to a depth-2 representation as described above?
2) Can a CBT represent the optimal policy for the task?
3) Assuming the existence of an optimal CBT, and given the left-to-right order of tick traversal, what is the impact on optimality of an arbitrary arrangement of behaviors at depth 2?

IV. Reduction to a CBT

Behavior trees, with their full range of features, are perhaps too powerful to be universally reducible to CBTs. However, we show in this section that many common and useful structures are indeed reducible to CBTs. We particularly exclude the parallel nodes which allow multiple actuators to operate simultaneously. Instead, we focus on systems that only allow one action (user defined behavior, or primitive action) to be active at any given tick.

One common feature in BTs is to organize condition nodes at multiple levels, exploiting the in-built AND-OR modality of BTs to combine these conditions. Figure 3 illustrates this case, and also shows how such a substructure can be reduced to a unit of the CBT.

Another common feature is a sequence node with memory (notated as an arrow followed by an asterisk), that allows

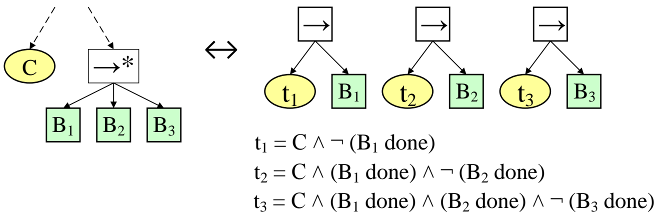


Fig. 4. A BT with memory sequence, and its unit reduction.

sequencing of multiple behaviors in a strict order, as shown in Figure 4. The memory feature allows the BT to remember which behavior was $running$ in the last tick, so that once B_1 is complete, successive ticks can be pushed directly to B_2 instead of restarting B_1. The condition C in Figure 4 (and in Figure 5 as well) represents the condition under which the memory sequence node receives a tick. It may not be a real node in the BT, but a condensation of all such conditions from the rest of the tree. Figure 4 also shows how such substructures can be reduced to units of the CBT. Note that this reduction requires additional tests regarding the status of the behaviors (done, or not done), that are expected to be reset after the sequence is complete. In our experiments, we emulate this reset operation by using a moving window over behavior completion with the $success$ status, instead of an explicit reset operation. Also note that the tests in Figure 4 could be simplified if we assumed that the units shown would be arranged successively in the CBT.

An obvious question at this point is what if the substructure is more complex, involving multiple behaviors at different levels, connected by sequences and selectors? Figure 5 shows a more elaborate example, with the corresponding reduction to units. It is clear that recording which behaviors completed with $success$ status (as indicated in the previous paragraph) alone is insufficient; we must also record which behaviors failed. Again, we could emulate this with the moving window over behavior terminations (with $success$ or $failure$). Furthermore, the set of predicates $\mathcal{P}$ need to be extended to include two additional predicates for each behavior's success or failure.

Similar to a memory sequence node described in Figure 4, there can also be a memory selector node, which would pass on a tick directly to B_2 if B_1 fails. These nodes are typically used to group together multiple ways of achieving the same goal, e.g., grabbing an object with one arm, vs. grabbing it with two arms. Multiple behaviors with the same goal add redundancy, which is an important technique to achieve robustness and graceful degradation. Clearly, memory selector nodes can be reduced in a similar way as memory sequence nodes, and implemented by recording $failure$ within the moving window as described in the previous paragraph.

Sometimes, a behavior may be used in multiple leaves in the same BT to exploit code reusability. In such cases, each occurrence may lead to a unit reduction using the strategies above, and the multiple resulting units can simply be unified with a single guard that is the disjunction of the individual unit guards. We shall show later that the order in which the units appear, and the other units that intervene, will not prevent such unification.

In concluding this section, we emphasize that we do *not* claim *every* behavior tree can be reduced to a CBT. Our claim is more modest—that many useful behavior trees can indeed be reduced to equivalent CBTs. More importantly, it is not actually necessary to reduce a BT to a CBT, rather the inverse transformation is of our interest, and this section illustrates the need to accommodate additional conditions (behavior done, or failed) in our CBT to enable an inverse transformation.

V. Optimal CBT

With any new policy language there is a naturally associated question whether or not the new policy representation is sufficiently expressive. One relevant measure is whether it can represent an optimal policy whenever it exists. We answer this question about the CBT in this section.

As in [16], a behavior can be viewed as a partition of the state space into subsets according to the return status function, ρ. More specifically, a behavior B partitions the state space Σ into disjoint subsets where it returns success (S) or failure (F) immediately, a subset where it executes a control action according to π_B and transitions to another state while returning $running$ status (R), or a subset where it has undefined behavior or is never supposed to be invoked (U, e.g., for features $f \notin \mathcal{F}_B$). Thus, at development time, every behavior B partitions Σ into $P_B = \{S_B, F_B, R_B, U_B\}$. This view of a behavior B can be shown to be equivalent to the definition of an option $O = \langle I_O, \pi_O, \beta_O \rangle$ as follows:

$$\sigma \in R_B \cup S_B \Leftrightarrow \sigma \in I_O$$
$$\sigma \in R_B \Leftrightarrow \pi_O(\sigma) = \pi_B(\sigma)$$
$$\sigma \in S_B \cup F_B \Leftrightarrow \beta_O(\sigma) = 1$$

Thus, every behavior corresponds to an option with deterministic termination—henceforth *deterministic option*, and every deterministic option corresponds to one or more behaviors. Note that the partition $P_B = \{S_B, F_B, R_B, U_B\}$ defined by a behavior B is only valid when it is created—a process usually independent of its guard or other behaviors. Once placed in a CBT, however, this partition changes as outlined below.

Proposition 1. *Every behavior B in a CBT is associated with a unique partition of Σ induced by the CBT.*

Proof: Assume $P_B = \{S_B, F_B, R_B, U_B\}$ before the behavior B is placed in the CBT, and $P'_B = \{S'_B, F'_B, R'_B, U'_B\}$ after the behavior is placed in the CBT, if such a partition exists. The CBT changes the partition P_B according to the following two rules $\forall \sigma \in \Sigma$:

$$\exists B'(B' \succ B) \wedge (\sigma \in S'_{B'} \cup R'_{B'} \cup F'_{B'}) \Rightarrow \sigma \in U'_B, \quad (3)$$
$$(g_B(\sigma) = fail) \Rightarrow \sigma \in U'_B, \quad (4)$$

where $\succ$ is the BT precedence in priority, and g_B stands for the guard of behavior B. In words, if there is a behavior of higher priority, B' (i.e., on the left of B in the CBT), that will pre-empt B in some states then the corresponding states

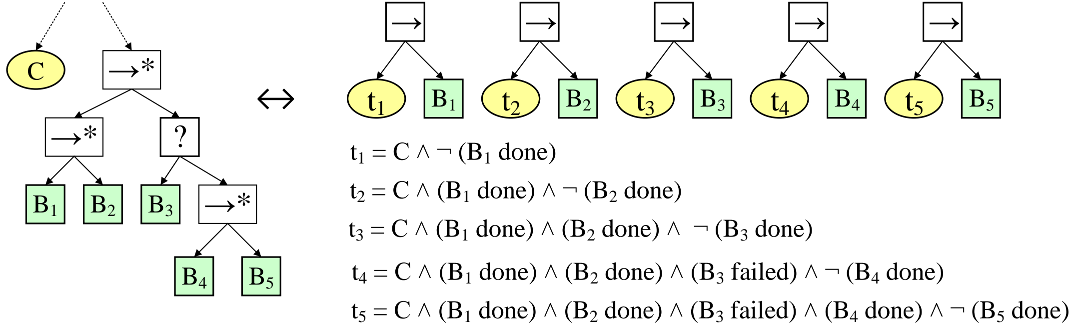


Fig. 5. A complex BT and its unit reduction.

are relocated to U'_B. Further, if the guard fails for a state, then it is also relocated to U'_B. Although multiple behaviors B' may satisfy rule (4), the leftmost of them will dominate. Thus the existence of a unique P'_B depends on the existence of unique partitions for behaviors to the left of B in the CBT. This argument extends inductively, until we reach the base case of the leftmost behavior in the CBT for which the partition is (potentially) only modified by rule (4), which does not depend on any other behavior. □

The following two observations follow directly from the above construction:

- **Observation 1:** for any state that is not in the U'_B subset of a behavior B in a CBT, B will receive a tick whenever the system is in that state.
- **Observation 2:** any state is outside the U' subset of no more than one behavior, since otherwise all but the leftmost of them would be usurped by rule (4). If we assume the inclusion of some behavior(s) that can be invoked in every possible state (i.e., *universal* behaviors), then we can strengthen the observation to: *any state is outside the U' subset of exactly one behavior*. Examples of such universal behaviors include idle/noop, ballistic turn, the wandering behavior in some settings, or even simply a primitive action (because they are special cases of options). Note that RL also returns exactly one action in any state via equation 2.

Given that each behavior corresponds to a deterministic option and vice versa, the above observations mean that every CBT corresponds to a (top-level) policy over options. Given the existence of an optimal policy [3] over options for any given set of options [15], it then follows that such a policy corresponds to at least one CBT. Note that the existence of an optimal CBT is independent of the question explored in the previous section, i.e., independent of whether or not the optimal CBT was the outcome of reduction from a BT. As stated earlier, we are more interested in the inverse conversion from a CBT (in particular, an optimal CBT) to a BT, described later in Section VII.

VI. Optimal Ordering of Behaviors

The above section established the existence of an optimal CBT, which clearly would designate a strict ordering of the behaviors, coupled with specific guards. It might appear that in order to learn the optimal CBT, one would need to learn both the guards and the behavior permutation. However, behavior permutation calls for a combinatorial search, which we show in this section is unnecessary.

The idea is that we could choose any permutation of the behaviors in the CBT, and still be able to reproduce the behavior of the optimal CBT by simply setting the guards in a certain way for the chosen permutation. We shall show in this section that for any permutation of behaviors in a CBT, there exists a set of guards for that CBT that emulates the optimal CBT. This would obviate the need to either know a priori, or learn, the behavior permutation of the optimal CBT.

We shall establish the result inductively, by showing that if such guards exist for k behaviors, then they also exist for $k+1$ behaviors, by constructing the guards inductively. Let us suppose that we choose the permutation of behaviors $B_1, \ldots, B_k$ in our CBT, with the corresponding guards $t_1^k, \ldots, t_k^k$ that are computed based on the k behaviors. In the optimal CBT, these behaviors may be permuted differently, and let the guards corresponding to $B_1, \ldots, B_k$ be $t'_1, \ldots, t'_k$. Now we insert a new behavior B_{k+1} with the guard t_{k+1} at the right end of our CBT, but in the optimal CBT it may be inserted (with guard t'_{k+1}) in a different location thereby shifting some behaviors rightward. Under this construction, the following holds:

Theorem 2. *If the guards $t_1^{k+1}, \ldots, t_{k+1}^{k+1}$ in our CBT are computed (on the basis of $k+1$ behaviors) recursively as*

$$t_i^{k+1} = \begin{cases} t_i^k & B_i \text{ not shifted in optimal CBT} \\ t_i^k \wedge \neg t'_{k+1} & i \in [1,k],\ B_i \text{ shifted} \\ t'_{k+1} & i = k+1 \end{cases} \tag{5}$$

then its behavior is identical to that of the optimal CBT of the $k+1$ behaviors.

Proof: For the base case, notice that when $k=1$, $t_1^1 = t'_1$, and our CBT will be equivalent to the optimal CBT. If B_2 is inserted to the right of B_1 in the optimal CBT, then $t_i^2 = t_i^1$, since no behavior is shifted (i.e., the optimal permutation is the same as in our CBT). However, if B_2 is inserted to the left of B_1 in the optimal CBT, then $t_1^2 = t'_1 \wedge \neg t'_2 = t_1^1 \wedge \neg t'_2$, while $t_2^2 = t'_2$. It is straightforward to verify that these settings of t_1^2 and t_2^2 will give equivalent behaviors for

[3] Note that an optimal policy over options may differ in performance from an optimal policy, unless the set of options includes the primitive actions.

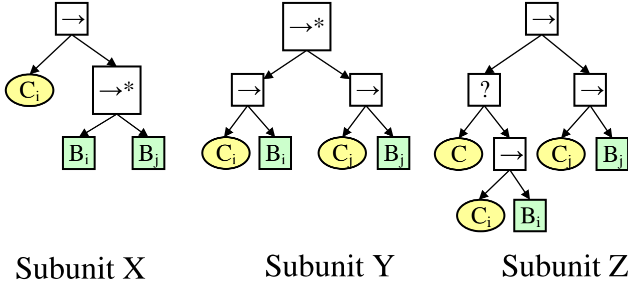


Fig. 6. The subunits referenced in Algorithm 1.

the two CBTs.

For the inductive case, we take the strong hypothesis that $t_1^k, \ldots, t_k^k$ can be computed on the basis of $t'_1, \ldots, t'_k$ and $t_1^{k-1}, \ldots, t_{k-1}^{k-1}$, to give equivalent behaviors of the two CBTs. Now assume that B_{k+1} is inserted somewhere in the optimal CBT. All behaviors to the left are unshifted, thus when they receive a tick will not be affected by this insertion. Hence their guards will need no change, proving $t_i^{k+1} = t_i^k$. Next consider B_{k+1} itself. It is a new behavior, therefore, its guard can be no different from t'_{k+1}. Finally, consider the behaviors that were shifted rightward. They will receive a tick in the optimal CBT not only when their previous guard t_i^k holds, but also when t'_{k+1} is false, allowing the tick to pass through the unit with B_{k+1}. This completes the proof. □

In the above proof we made use of a strong hypothesis to account for the subtlety that when a new behavior B_{k+1} is inserted into an optimal CBT containing k behaviors, the relative permutation of the existing k behaviors may also need to change. Thus the construction rules in equation 5 should be interpreted as being applied *after* the changed $t_1^k, \ldots, t_k^k$ have been computed.

Note that rules in equation 5 cannot actually be computed without knowing $t'_1, \ldots, t'_{k+1}$ from the optimal CBT. However, the result does imply that such guards $t_1^{k+1}, \ldots, t_{k+1}^{k+1}$ *exist*. Together with Observation 2's (in previous section) consistency with RL, this Theorem implies that the computation of the guards for the behaviors can be deferred until *after* RL has generated the optimal CBT. The guards for this optimal CBT can be extracted directly from RL's output (explained in the next section) for any chosen permutation of the behaviors.

VII. De-Canonicalization

As mentioned in Section II-B, here we assume a simple tabular form of RL that yields a direct mapping from $\prod_i \mathcal{P}_i$ to the optimal behaviors. Only the predicate combinations (minterms/products, i.e., conjunctions of all predicates or their negations) seen during RL are in the map, while the unseen combinations are assumed to be "don't care"s, i.e. they could be mapped to any behavior. We collect all the minterms for each behavior B_i from the map, and call this set $minterms(B_i)$. Logic simplification can minimize $minterms(B_i)$ into compact guards t_i. The set of pairs (t_i, B_i) yields a learned CBT. On the one hand, the learned guards t_i may still be less readable logic expressions. On the other, it may be useful (again for the sake of human readability) to replace recognizable structures (such as memory sequence/selector) within the flattened CBT representation, in order to transform the learned CBT into a BT from which it could have been created by our canonicalization procedure as illustrated in Section IV. Thus, we are interested in an operation that is inverse to the canonicalization steps, that also invokes logic simplification. We call this operation *de-canonicalization*, and the resulting BT the *decanonicalized BT*.

Algorithm 1 `De-Canonicalize`

Input: A set of (t_i, B_i)-pairs, $minterms(B_i)$ and the set $dontcares$ of unseen predicate combinations.

1: **repeat**
2:   **repeat**
3:     **if** $\exists B_i, B_j, c$, s.t. $t_i = c_i \land \neg c, t_j = c_j \land c$ **then**
4:       **if** $c = B_i done$ **then**
5:         **if** $c_i = c_j$ **then**
6:           Substitute subunit X (Fig. 6) for t_i, B_i, t_j, B_j.
7:         **else**
8:           Substitute subunit Y for t_i, B_i, t_j, B_j.
9:         **end if**
10:       **else**
11:         Substitute subunit Z for t_i, B_i, t_j, B_j.
12:       **end if**
13:       $minterms(X/Y/Z) \leftarrow minterms(B_i) \cup minterms(B_j)$
14:     **end if**
15:   **until** No further B_i, B_j, c
16:   Make remaining behaviors separate subunits, and order them by increasing length of t_i.
17:   **for** subunit X_k in the chosen order **do**
18:     $t_k \leftarrow$`LogicMin`$(minterms(X_k), dontcares)$
19:     $dontcares \leftarrow dontcares \cup minterms(X_k)$
20:   **end for**
21: **until** No change
22: Place all subunits under a selector node and return the de-canonicalized BT.

Algorithm 1 shows de-canonicalization, using inputs (t_i, B_i)-pairs, $minterms(B_i)$ and $dontcares$, as described above. In lines 2–9, Subunit X (Figure 6) is used for memory sequences that have a common guard, but Y for those that have different guards for the behaviors in a memory sequence. An analogous **if**-block can be added to identify memory selectors, but is omitted here because our evaluation domains (see next section) did not use behaviors that can fail after successful initiation. Algorithm 1 can only build memory sequences of length 2, because we maintained moving window over just the last successful behavior. With longer windows, the algorithm can be extended to build longer memory sequences. Subunit Z is used for simplifying guards that can help identify additional pairs in future passes of the

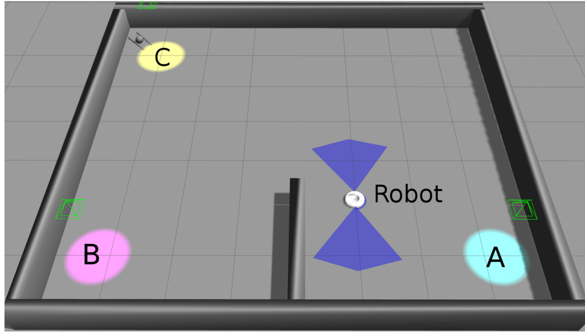


Fig. 7. The package delivery domain in Gazebo.

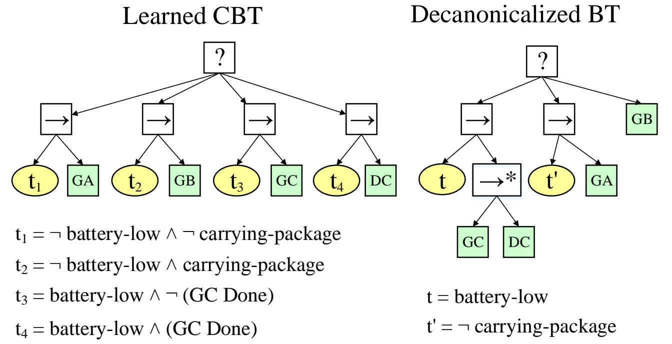


Fig. 8. The learned optimal policy in the package delivery domain.

repeat-loop 1–21. The replacement of recognizable subunits by X,Y, or Z is an instance of *graph rewriting*. Line 18 invokes a logic minimization procedure–`LogicMin`, while line 19 essentially imposes the left-to-right priority in guard simplification, which ultimately leads to the rightmost behavior having the guard `True`. This is why we order the longest guard last (line 16). Although most of Algorithm 1 involves polynomial steps, unfortunately logic minimization is NP-complete. We use a popular efficient heuristic algorithm, Espresso [17], in particular its implementation in PyEDA, for `LogicMin`.

VIII. Evaluation

We use a package delivery domain for evaluation, where an iRobot Create 2 picks up packages at location A and drops them off at location B repeatedly, except to take a recharging detour (to a charger at location C) when it senses low battery, as shown in Figure 7. Package load and unload operations occur automatically when the robot reaches A and B respectively, but to keep the domain simple we do not actually implement these operations; the robot simply pretends to load or unload. We use two variants of the domain, a smaller version with no obstacles, and a larger version that contains the barrier shown in Figure 7 along with more behaviors and predicates. Next we present the details of the tasks, and show the results of RL (learned CBT) and Algorithm 1 (de-canonicalized BT) for both variants of package delivery.

A. Package Delivery Domain: Small

In the small variant, the robot's set of behaviors includes Goto-A (GA), Goto-B (GB), Goto-C (GC), and Dock-Charger (DC). The last behavior allows the robot to dock its charging pins correctly with a wall charger's, and can only be executed when the robot is at location C. The completion of the Dock-Charger behavior fully recharges the robot's battery instantly. Without recharging, the battery drains at the rate of 0.1% per tick. There is also a directional IR beacon between the charger's pins, to help the robot orient, which can only be sensed from within the circular region highlighted around C. In the small version of this domain, the barrier between A and B is absent, and the robot does not have the side laser scanners shown in Figure 7. The robot uses multiple sensors to be able to navigate the domain, viz., front bump sensors to detect wall collisions, and short range IR sensors to detect the charger beacon and adjust the approach direction for the Dock-Charger behavior. Additionally, it knows the coordinates of A, B, C and also senses its current location and orientation. These sensor features define Σ on the basis of which the 4 behaviors are hand-coded. The given set of predicates is $\mathcal{P} = \{carrying_package, battery_low\}$, where the latter is true at and below 25% charge. To this we add 4 more predicates, each corresponding to *success* of a behavior, i.e., B1-done,..., B4-done. This emulates the moving window over the last successful behavior. Because of the way the behaviors are coded, they cannot fail once successfully invoked, i.e., they can fail outright (if invoked in states in $F_B \cup U_B$), but not afterwards. Thus we do not need behavior failure predicates for this paper, but this is an important direction for future extension. For tabular Q-learning, we treat a change in any of the 6 predicates as a state transition. The reward function is: reward = +1 when Goto-A completes successfully (i.e., package picked up), +2 when Goto-B completes successfully (i.e., package dropped off), -5 if episode terminates (i.e., dead battery), and -0.1 for all other cases. The behaviors were implemented (as servo) separately from RL, all in C++ within ROS and the Gazebo simulator. RL yields a map of 10 predicate combinations (minterms) to learned optimal behaviors $\{B_1, \ldots, B_4\}$, while the remaining $2^6 - 10$ combinations (unseen because they are impossible, e.g., DC-done and battery_low) form the $dontcares$. Figure 8 (left) shows the learned CBT, and the result of Algorithm 1 is on the right. A video of the learned policy is included as a supplement.

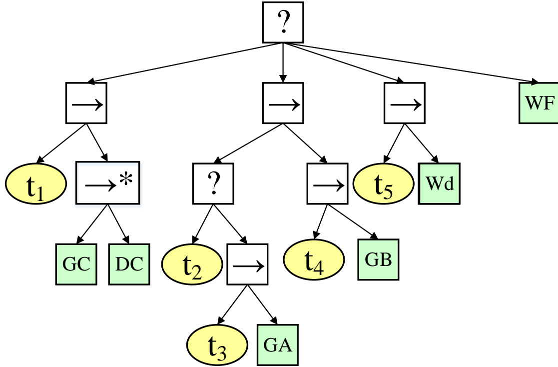
B. Package Delivery Domain: Large

Figure 7 shows the full version of the package delivery domain, where we add a barrier between A and B in the environment, and add two more behaviors–Wall-Follow (WF) and Wander (Wd). We equip the robot with laser scanners (to sense walls while wall-following) on both sides, and add 3 visibility predicates, $vis(A|B|C)$, based on line of sight. Together with a predicate for the successful completion of each of the 6 behaviors, we thus have 11 predicates for RL. In this domain, RL yields a map of 61 predicate combinations (minterms) to learned optimal behaviors, leaving the remain-

B_i	t_i
GA	$V_A \wedge \overline{C_P} \wedge \overline{L_B}$
GB	$V_B \wedge C_P \wedge \overline{L_B}$
GC	$\overline{GC\ Done} \wedge V_C \wedge L_B$
DC	$(GC\ Done) \wedge V_C \wedge L_B$
WF	$(\overline{V_B} \wedge C_P \wedge \overline{L_B}) \vee (\overline{V_C} \wedge (L_B \vee C_P)) \vee (\overline{V_A} \wedge \overline{C_P} \wedge \overline{L_B})$
Wd	$V_C \wedge \overline{V_B} \wedge C_P \wedge \overline{L_B}$

TABLE I

THE LEARNED TESTS (LEARNED CBT) FOR LARGE PACKAGE DELIVERY.



t_1 = battery-low ∧ vis(C) t_4 = vis(B)

t_2 = carrying-package t_5 = carrying-package ∧ vis(C)

t_3 = vis(A)

Fig. 9. The (de-canonicalized) optimal policy in the full package delivery domain.

ing $2^{11} - 61$ unseen combinations as dontcares. The resulting learned CBT is shown in Table I. This table uses shorthands V_X for $vis(X)$, C_P for $carrying_package$ and L_B for $battery_low$. The result of Algorithm 1 is shown in Figure 9. Notice that the replacement of intractable `LogicMin` by efficient heuristic (Espresso) has not affected the accuracy or readability of the result.

IX. CONCLUSIONS AND FUTURE DIRECTIONS

We have presented a first step in the autonomous acquisition of a behavior tree controller for a robot by leveraging reinforcement learning, with behavior coding being the only user input. We have proven the soundness of delayed commitment to behavior priority ordering, while preserving optimality. Two versions of a package delivery domain illustrated not only how our approach can produce readable controllers, but also that the replacement of an NP-complete step by an efficient heuristic may not affect the accuracy or the readability of the result. Several future extensions are possible, including accommodation of parallel and decorator nodes, behavior failures, and additional human intuitable patterns in de-canonicalization, including longer memory selector/sequences.

The behavior tree as a language for expressing plans/controllers also has untapped potential in multi-robot systems [18], both in the context of computing/learning coordinated plans for robot teammates [19], as well as constructing plan libraries for application in plan recognition [20]. The need for uniform action durations in both planning and plan recognition has been a significant constraint when it comes to practical applications particularly in multi-agent/robot systems, and behavior trees offers an elegant solution, thus opening up exciting future directions.

X. ACKNOWLEDGMENT

The author gratefully acknowledges constructive feedback from anonymous reviewers. This work was supported in part by a National Science Foundation grant IIS-1526813.

REFERENCES

[1] R. Parr and S. Russell, “Reinforcement learning with hierarchies of machines,” in *Proc. NIPS*, 1997.

[2] N. Meuleau, L. Peshkin, K. Kim, and L. Kaelbling, “Learning finite-state controllers for partially observable environments,” in *Proc. UAI*, 1999, pp. 427–436.

[3] D. Isla, “Handling complexity in the Halo 2 AI,” in *Game Developers Conference*, 2005.

[4] N. J. Nilsson, “Teleo-reactive programs for agent control,” *JAIR*, vol. 1, pp. 139–158, 1994.

[5] R. Arkin, *Behavior Based Robotics*. The MIT Press, 1998.

[6] R. A. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal on Robotics and Automation*, vol. RA-2, no. 1, pp. 14–22, 1986.

[7] J. A. Bagnell, F. Cavalcanti, T. G. L. Cui, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois, and R. Zhu, “An integrated system for autonomous robotics manipulation,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 2955–2962.

[8] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, “Towards a unified behavior trees framework for robot control,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[9] M. Colledanchise, “Behavior trees in robotics,” Ph.D. dissertation, KTH Royal Institute of Technology, Sweden, 2017.

[10] M. Colledanchise, R. Parasuraman, and P. Ögren, “Learning of behavior trees for autonomous agents,” *CoRR*, vol. abs/1504.05811, 2015. [Online]. Available: http://arxiv.org/abs/1504.05811

[11] 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)*, 2013, pp. 1–8.

[12] R. de Pontes Pereira and P. M. Engel, “A framework for constrained and adaptive behavior-based agents,” *CoRR*, vol. abs/1506.02312, 2015. [Online]. Available: http://arxiv.org/abs/1506.02312

[13] G. Robertson and I. Watson, “Building behavior trees from observations in real-time strategy games,” in *2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA)*, 2015, pp. 1–7.

[14] R. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[15] R. Sutton, D. Precup, and S. Singh, “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, vol. 112, pp. 181–211, 1999.

[16] M. Colledanchise and P. Ögren, “How behavior trees modularize robustness and safety in hybrid systems,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS 2014) :*, 2014, pp. 1482–1488.

[17] R. Brayton, G. Hachtel, C. McMullen, S. Vincentelli, and A. Luigi, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[18] T.-C. Au, B. Banerjee, P. Dasgupta, and P. Stone, “Multirobot systems,” *IEEE Intelligent Systems (Guest Editorial)*, vol. 32, no. 6, pp. 3–5, 2017.

[19] L. Kraemer and B. Banerjee, “Multi-agent reinforcement learning as a rehearsal for decentralized planning,” *Neurocomputing*, vol. 190, pp. 82–94, 2016.

[20] B. Banerjee, J. Lyle, and L. Kraemer, “The complexity of multi-agent plan recognition,” *Autonomous Agents and Multi-Agent Systems*, vol. 29, no. 1, pp. 40–72, 2015.