Task Planning with Belief Behavior Trees

Preprint · August 2020

CITATIONS
0

READS
12

3 authors, including:

Michele Colledanchise
Istituto Italiano di Tecnologia
27 PUBLICATIONS **345** CITATIONS
SEE PROFILE

Lorenzo Natale
Istituto Italiano di Tecnologia
225 PUBLICATIONS **5,467** CITATIONS
SEE PROFILE

Task Planning with Belief Behavior Trees

Evgenii Safronov[1,2], Michele Colledanchise[1], and Lorenzo Natale[1]

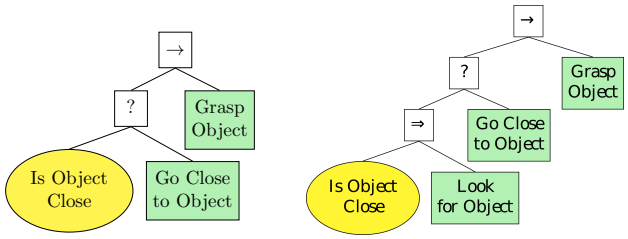
***Abstract*— In this paper, we propose *Belief Behavior Trees* (BBTs), an extension to Behavior Trees (BTs) that allows to automatically create a policy that controls a robot in partially observable environments. We extend the semantic of BTs to account for the uncertainty that affects both the conditions and action nodes of the BT. The tree gets synthesized following a planning strategy for BTs proposed recently: from a set of goal conditions we iteratively select a goal and find the action, or in general the subtree, that satisfies it. Such action may have preconditions that do not hold. For those preconditions, we find an action or subtree in the same fashion. We extend this approach by including, in the planner, actions that have the purpose to reduce the uncertainty that affects the value of a condition node in the BT (for example, turning on the lights to have better lighting conditions). We demonstrate that *BBTs* allows task planning with non-deterministic outcomes for actions. We provide experimental validation of our approach in a real robotic scenario and – for sake of reproducibility – in a simulated one.**

I. Introduction

The video game industry proposed Behavior Trees (BTs) as an alternative to statecharts to describe the behavior of non-playable characters. The industry found BTs successful to the point that they became an established tool appearing in textbooks [1]–[3] and game-coding software such as Pygame, Craft AI, and Unreal Engine. In robotics, both academia and industry follow a similar trend and use BTs to describe complex behaviors in a compact way. Moreover, BTs generalize other successful control architectures such as statecharts, the Subsumption architecture [**?**], and the Teleo-reactive Paradigm [3].

The particular syntax and semantic of BTs, which will be described later, allow a BT to continuously check a set of conditions to evaluate the actions to be performed and the ones to be aborted, if any. However, in the classical semantic of BT, the designer assumes that the values of such conditions are either true or false. While this appears like a natural way to describe policies, it underlies an assumption: the conditions involve observable variables. Consider the example in Figure 1a, the BT encodes the behavior that can be verbally described: as *whenever the robot is close to the object, grasp it, when the robot is not close to the object, go close to it*. Such behavior makes sense as long as the robot can observe the object position, so that the BT can evaluate the condition *object close*.

The real world remains intrinsically non-observable. However, in some cases, some variables can be observed, after performing specific actions or, in general, on a specific subspace in the state space. For example, the condition *object close* becomes observable only after the robot finds the object or if the robot knows the object position a priori.



(a) BT with classic semantic of condition nodes.

(b) BT with new semantic of condition nodes.

Fig. 1. Examples of BTs using the classic (left) and the new (right) semantic for conditions. The syntax and semantic are described later in this paper.

In this paper, following the recent advances of BT semantic [4], we allow the conditions to be either true, false, or *unknown*. Doing so, we can construct a new BT (in Figure 1b) that, in addition to the functionalities of the original BT, *it performs a action that looks for the object whenever the value of its position relative to the robot is unknown*. Inspired by our recent work [5], a task planner generates such BT. The resulting BT performs both *actuation*, which change the state space to achieve the goal and *perception*, which makes observations on the robot's space. For the variables that are latent, the task planner operates directly on the *belief state*, a state-space of the probability distribution over physical states.

Given the initial physical state of the system (with possibly unobserved conditions) and a goal definition, the algorithm refines the tree until it results in successful execution. Having moved from physical to *belief* states, on each planning step we are looking for the *most probable* unsatisfied condition. If the condition holds *false*, we insert an *actuation* action, if it is *unobserved*, we insert a *perception* action. As both actuations and perceptions have probabilistic outcomes, we developed an extention of standard BTs, which we called *Belief Behavior Trees* (BBTs). BBTs allows to perform planning in the belief space, by applying mulitple ticks, therefore simulating all possible scenarios of execution and computing the *probability of success* of each execution trace.

To summarize, in this paper we combine recent advances in the BT community to take a step towards BT planning in partially-observable environments. We show how to handle uncertainty in the BT formulation, then propose the *BBT*, a BT that allows planning with the non-deterministic out-

[1]The authors are with Istituto Italiano di Tecnologia, Genoa, Italy.
[2]Evgenii Safronov is also with Department of Informatics, Bioengineering, Robotics and Systems Engineering, Università di Genova, Genova, Italy evgenii.safronov@iit.it

comes for actions and conditions. We provide experimental validation running our approach in a real robotic scenario and on a simulation. For reproducibility, we also make available online the source code of our framework and the simulated scenario.

II. Related Works

The solution to long-horizon task planning in uncertain environments lies beyond the state of the art [6], [7]. Early works apply the *determinize-and-plan* approach where they compute the most likely physical state and then plan in a deterministic domain from that state. However, such approaches have a fundamental issue, they cannot consider actions that reduce the uncertainty as the planner runs on a deterministic domain. Later works include perception actions in the planner, however, they assume that the future observations are always the most probable ones [8]. Other works [9], extend the classical planning operators [7] defining preconditions and effects in the belief space to use off-the-shelf task planners to operate in belief space. However, they still rely on the maximum likelihood observation assumption above. Kaelbling et al. [6] outlined a framework that blends acting and planning to handle uncertainty in robot tasks. They construct task plans in belief space under maximum likelihood observation assumption.

Regarding the automatic creation of BTs, the research community follows two main directions: a learning one, where they construct the BTs in a model-free fashion optimizing over a reward function, and a task planning one, where they construct the BTs in a model-based fashion. Works include the use of Reinforcement Learning techniques [10] and in particular genetic programming to build and combine BTs selecting the ones that have the highest reward [11]. Other works [12] construct the BT by mixing a greedy approach with genetic programming to maximize the reward function. Recent works combine the manual design of BTs with machine learning techniques to learn policies with the desired performance [13]. Other works synthesize BTs from demonstration [14], [15]. The employment of task planning approaches is a recent trend, early works define a systematic framework to automatically generate complex BT structures from a repository of simpler ones. However, that approach strongly depends on a repository of hand-made structures (in a similar fashion of Hierarchical Task Network), whereas our approach automatically creates BTs from a set of simple actions and their pre- and postconditions. Other works define the goal in the form of a Linear Temporal Logic specification and then employ a verification tool to synthesize a BT that is correct by construction [16]. Later works propose the construction of the BT based on the idea of backchaining. Starting from the goal condition the algorithm finds actions that meet those conditions [5].

The fundamental difference with all the work above on automatic synthesis of BTs lies in the fact that we do consider uncertainty in both the actions outcome (i.e. actuation and perception) and the value of condition nodes.

III. Background

In this section, we briefly describe the syntax and semantic of BTs. We follow conventional BT syntax with few important differences. The literature includes detailed descriptions [3], [4].

We use the common definitions of *parent* and *child* for the tree structure. The root is the node without parents, the nodes have only one parent. Graphically, the children of nodes are placed below it, as shown in Figure 1. The children are executed in the order from left to right.

The execution of a BT begins from the root node. It sends *ticks* [1] its children, from left to right. When a parent sends a tick to a child, the child can be executed. The child returns to the parent a status *running* ($\mathbb{R}$) if its execution has not finished yet, *success* ($\mathbb{S}$) if it has achieved its goal, or *failure* ($\mathbb{F}$) otherwise.

We describe the semantic of the nodes we use in this paper.

Fallback: The fallback is a control node that returns status success $\mathbb{S}$ (running $\mathbb{R}$) as soon as it finds a child that returns success $\mathbb{S}$ (running $\mathbb{R}$). It returns failure $\mathbb{F}$ only if all the children return failure $\mathbb{F}$. When a child returns running or success, the fallback node does not tick the next child (if any). The fallback node is represented by a box with a "?", as in Figure 1.

Sequence: The sequence node returns failure (running) as soon as it finds a child that returns failure (running). It returns success only if all the children return success. When a child returns running or failure, the sequence node does not tick the next child (if any). The sequence node is represented by a box with a "$\rightarrow$", as in Figure 1.

Skipper: The skipper node return success (failure) if a child return success (failure) otherwise it ticks the next child. The skipper node is represented by a box with a "$\Rightarrow$", as in Figure 1.

Action: The action node returns success if the action is completed and failure if the action cannot be completed. Otherwise, it returns running. An action node is represented as a rectangle, as in Figure 1.

Condition: The condition node checks if a condition is satisfied or not, returning success or failure accordingly. In this paper we allow conditions to return running to encode a condition whose value is unknown, as described in Section I. A condition node is represented as an ellipse, as in Figure 1.

IV. Problem Formulation

Conditions are defined by their literal, e.g., $luminousity\_ok$ may hold a value from the set of $\{true,\ false,\ unknown\}$. Actions are described by their set of preconditions and action outcomes. The set of preconditions C^a_{pre} for the action a is a set of (condition, value) pairs. An action could be only executed if all its preconditions hold. Action outcomes are probabilistic distribution over postcondition sets, i.e. for action a we can have[2] $\bigcup_i p_i C^a_{post_i}$. The action execution results in a change

[1] A tick is a signal that allows the execution of a child.

[2] Here and after $\bigcup_i p_i A_i$ denotes a discrete probability distribution of A with probabilities p_i.

of some condition variables. Hence the problem can be stated as follows: given a set of actions defined above and a set of goal conditions, define a task planning algorithm that builds a BT policy that satisfies the goal conditions with no less than target probability.

V. Handling Uncertainty in Behavior Trees

In this section, we present the first contribution of this paper. We show how to handle two types of uncertainty in the BT formulation: the *current state uncertainty* and the *future state uncertainty*.

A. Handling current state uncertainty in BTs

In many real-world scenarios, the robot only partially observes the environment. Geometrical occlusions (e.g., doors, furniture, other objects), poor signal-to-noise ratio (e.g., low luminosity for the vision and high noise level for sound recording), and the sensor noise cause the current state uncertainty.

From a task planning standpoint, a set of conditions describes the current state. Such conditions take values either *true* or *false*. These conditions correspond to condition nodes of BT, which return respectively *success* or *failure*. However, in the case of partially-observable environments, the value of a condition can be unknown. Looking back at the example in Figure 1, if we allow conditions in the BT to return *running* status whenever the value of the condition is unknown, then we can use the Skipper node [4] to execute the observation. Such an observation could be implemented in form of a single action or as a subtree of BT.

B. Handling future state uncertainty in BTs

An action could result in different possible outcomes. For example, an attempt to grasp a bottle from a table could either result in a successful grasp, in a minor failure (e.g. the bottle stays on a table state of the world is unchanged), or in a severe failure (e.g. the bottle falls on the floor).

In this paper, we describe actions along with the *preconditions* (i.e., the set of conditions that must be true before their execution) and the *postconditions* (i.e., the set of conditions that must be true after the action terminates).

To account for future state uncertainty, we define a set of all the possible postconditions. Assuming that the possible postconditions are independent events, we can add a probability of each postconditions set and define action outcome as a probabilistic sum:

$$\hat{a} := \{C^a_{pre}, C^a_{post}\} \longrightarrow \hat{a} := \{C^a_{pre}, \bigcup_i p^a_i C^a_{post_i}\}$$
$$C^a_{pre} \subset C, C^a_{post} \subset C, C^a_{post_i} \subset C \quad (1)$$

where C is a set of all conditions, C^a_{pre}, is the set of precondition, and C^a_{post} is the set of postconditions. In this work, we do not discuss the origin of these probabilities. They can be learned automatically during the task execution or calculated from robot safety-related and performance parameters.

C. Uncertainty entanglement

It is important to highlight that current and future state uncertainties are *not* independent in the general case. We need to point out cases when current state uncertainty leads to future state uncertainty and vice versa.

Perception actions: Sometimes a robot can make an extra action to resolve a current state uncertainty. For example, it might open a fridge to check if $in\_fridge(soda)$ is true. It makes sense to perform this action only in case if the value of condition above is $unknown$. Hence, observations transform current state uncertainty into future state uncertainty.

	Preconditions	Postconditions
look for object in fridge	$in\_fridge(obj) = \mathbb{R}$	$0.5, in\_fridge(obj) = \mathbb{S}$ $0.5, in\_fridge(obj) = \mathbb{F}$
grasp	...	$1, grasped(obj) = \mathbb{R}$
after_grasp_check	$grasped(obj) = \mathbb{R}$	$0.8, grasped(obj) = \mathbb{F}$ $0.2, grasped(obj) = \mathbb{S}$

TABLE I. Uncertainty entanglement examples.

Actions with unknown outcomes: Another important example is the action *grasp*. For some robots, it is possible to immediately sense if the object was successfully grasped (e.g., by tactile sensors). For others, the result of grasping is *unknown* until we make an *observation* (e.g., through putting the object close to the robot's camera for correct recognition). In this case, the action *grasp* can have only one postcondition, but with *unknown* value[3] (see Table I).

Note, that if our grasping performs well, the success probability of $after\_grasp\_check$ observation should be significantly higher than the one for an arbitrary observation. To fix that, one can use additional postcondition $after\_grasp = \mathbb{S}$ for $grasp$ and use it as a precondition for $grasp\_check$ to enforce this type of observation. Another way to solve it is to put both actions (actuation and sensing) together in a sequence making a sub-BT or node template, to let the robot always make an observation after grasping.

VI. Belief Behavior Tree

In this section, we present the second contribution of this paper. We define BBTs, an extension to the definition of BT where conditions return running whenever the value of the condition is not known and actions have nondeterministic execution outcome. The planning will result in the construction of BBT.

Belief state in BTs: A physical state contains all conditions and their values. Belief state m is defined as a probabilistic distribution of physical states s_i as follows.

$$m := \bigcup_i p_i s_i$$

where p_i is the probability of being in state s_i.

A. Leaf nodes

We now describe how actions and conditions and their corresponding BBT nodes act on a belief state.

[3]Recall that logic states true, false or unknown are mapped to success (S), failure (F) and running (R) respectively in the BT

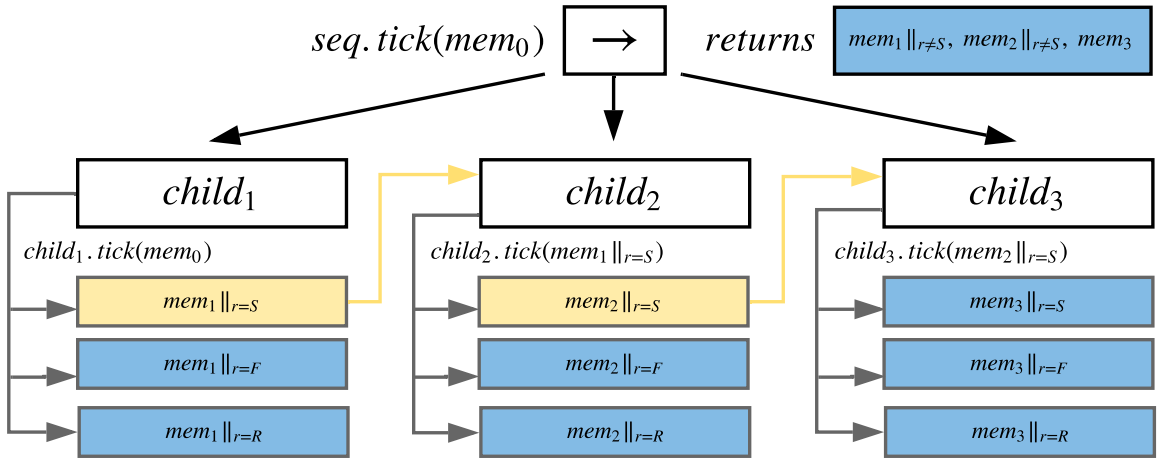


Fig. 2. A belief state flow chart example for a tick execution in a Sequence control node. Physical states highlighted in blue forms the returned belief state by a sequence.

Actions: Deterministic actions modifies a physical state:

$$s' = \hat{a}\,(s)$$

The action's effect on a belief state could be defined intuitively:

$$m' = \hat{a}\,(m) := \bigcup_i p_i \hat{a}\,(s_i)$$

Actions $\hat{A}$ with probabilistic outcomes $\hat{a}_j$ applied to a physical state s result in a belief state:

$$m' = \hat{A}\,(s) := \bigcup_j p_j \hat{a}_j\,(s)$$

And a result of an action with probabilistic outcomes, applied to a belief state is also a belief state:

$$m' = \hat{A}\,(m) = \bigcup_i p_i \bigcup_j p_j \hat{a}_j\,(s_i) = \bigcup_{i,j} p_i p_j \hat{a}_j\,(s_i)$$

If we define an action by its postconditions (as in Eq. 1), then setting postcondition values fully describes $\hat{a}_j$ functions. Corresponding Action node in BBT acts on a belief state as described above and returns a *success* status. In the classical BT formulation, a node returns only the status to its parent. In BBTs, a node returns a full belief state and includes the returned status as variable r_i to the belief state:

$$\hat{A}\,(m) = \bigcup_{i,j} p_i p_j \{f_j(s_i) | r_i := \mathbb{S}\} \tag{2}$$

Conditions: Conditions are defined as a function $c(s)$ over a physical state which returns one of three statuses ($\mathbb{S}$, $\mathbb{F}$, and $\mathbb{R}$). Condition node can be perceived as an action node that modifies only return status r variable of each physical state.

$$C\,(m) = \bigcup_i p_i \{s_i | r_i := \hat{c}\,(s_i)\}$$

B. Control nodes

Control node execution starts from a tick, applied to the first child. Working in belief space, we cannot return single status, as for different physical state there might be different execution results. Hence, each node in the BBT returns a belief state instead of a single status ($\mathbb{S}$, $\mathbb{F}$ or $\mathbb{R}$). For each physical state in returned belief state, we save returned status in variable r_i (i denotes iteration over physical states). In a BT, further execution of the control node's children depends on a returned status and therefore can vary among physical states. As it was noted in the previous work [4], Control nodes (Sequence, Fallback, and Skipper) rely on the same execution algorithm up to the *return status* parameter. Hence, consider a Sequence node with all children being leaves. Whenever the Sequence node receives a status of $\mathbb{F}$ or $\mathbb{R}$, it returns such state to its parent. Given that, in BBT for all physical states returned by the first child whether the $r_i = \mathbb{S}$, we should continue execution. Executing a Sequence node the BBT, we select all the physical states whether $r_i = \mathbb{S}$, and pass them as an argument to the tick of second child. All the remaining physical states with $r_i \neq \mathbb{S}$, should not be passed but returned to the parent. We repeat the same procedure for all the children, passing a subset with $r_i = \mathbb{S}$ to the next child and collecting all the other physical states for return. The whole returned belief state of the last child should be added up to the return of a Sequential control node.
Formally, for the recursive definition of the tick function we have to add two more arguments:

- `mem` - to pass a subset of Belief State
- `from` - to apply a tick from a *i-th* child

Hence, we can obtain an elegant and compact definition of belief tick (Alg. 1). The tick function is now recursive not only with depth of the BT, but also in the left to right direction of the control node children. Therefore, it

contains second argument *from* that denotes the certain child of control node. If we the belief state argument is empty set, we return the empty set. In case we reached end of node's children list, we return the belief state argument. In other cases, we apply tick to child at the position *from* in the node's children list. The result of tick is a belief state, that might contains different return statuses. Hence, we separate the physical states where the child node returned $\mathbb{S}$status from the physical state where the child returned $\mathbb{F}$or $\mathbb{R}$. In the latter case Sequence node terminates the execution, so we add these states to returned belief state. $\mathbb{S}$part of belief state we pass to the next child of the Sequence. Notice that an example of belief state flow chart is reported in Fig. 2. Calling a tick function on the root node with current

Algorithm 1 Tick function for Sequential

```
1:  function NODE.TICK(mem, from = 0)
2:      if from ≥ len(node.children) then
3:          return mem
4:      end if
5:      if mem = ∅ then
6:          return ∅
7:      end if
8:      mem = node.children[from].tick(mem)
9:      succeeded, failed = mem.split_by(r_i = S)
10:     return failed + node.tick(succeeded, from + 1)
11: end function
```

belief state as an argument, the resulting belief state will fully describe all possible scenarios of tick propagation on all initial physical states.

C. Delayed action outcomes

In general, actions may imply that it takes some time to change the system's state while the tick execution in most works is a non-blocking procedure [17]. If we directly apply actions in the way defined in Eq. 2, we would fork our belief state immediately before executing further BT nodes. That could result in a different behavior, compared to real BT execution. To avoid potential inconsistency, we *delay* action outcomes by assigning a link to a delayed action to a specific state variable. If we never execute two or more actions in parallel, then the execution shall wait until action was finished. However, as we do not execute any action during this delay, we can assume for the simulation that an action finishes after one tick of the BT. So, we apply the action outcomes to belief state right before next tick. If the BT executes multiple actions in one tick e.g., by a Parallel node [3], we could have multiple delayed actions simultaneously. Then, applying these actions on a belief state in a different order and with a different number of ticks in between could result in a different execution. Handling *multiple* delayed actions and a Parallel node is out of the scope of this paper.

D. Self-simulation

We now define a *self-simulation* procedure. Informally, it could be described as a forward belief statespace exploration

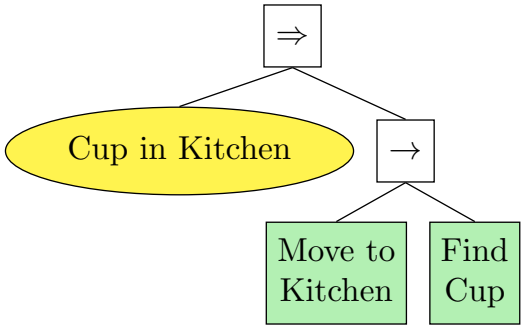


Fig. 3. An extra sequence attached to the Skipper node imposes the desired behavior where the robot first moves to the kitchen and then finds the cup.

procedure from an initial state applying a BBT policy.

Giving the initial belief state and a BBT, we can simulate all the scenarios of execution. Note that the initial belief state could be simply a physical state with the probability of 1 representing the current state of the robot. We define the self-simulation procedure limited to the cases when a robot could perform only one action at a time. For this reasons, we do not use Parallel node in this work, and always attach Action nodes to the Skipper through extra Sequence node to prevent tick passing to the further children (see the example on Fig. 3). So, we can apply a delayed action right after root tick finishes. Applying a tick to the root node and applying delayed action to the returned memory form a step of BBT *self-simulation*. We can set a limit on number of ticks or the maximum number of states in a memory (the latter corresponds to the number of scenarios simulated). Notice that as we do not have any other source of memory changes, except for actions, and if for some physical state we do not have any delayed actions, this physical state shall be unchanged by further execution. Being executed on the same physical state, BT is guaranteed to reproduce the same tree traversal, and therefore, no actions shall be executed. Such physical states could be excluded from a belief state passed to the next root tick. Moreover, if all our actions are latched (i.e. once the action is finished, it shall not be executed again but return last status, $\mathbb{S}$or $\mathbb{F}$), the number of action nodes calls is limited to the number of actions in the tree. In this case, BT is guaranteed to finish execution in a finite number of ticks.

Algorithm 2 Self-simulation procedure

```
1:  function BT.SIMULATE(mem)
2:      results = BeliefState([])
3:      while not mem.empty() do
4:          current = BT.root.tick(mem)
5:          ended, mem = current.split_by(
6:           s : s.has_delayed_actions())
7:          results = results + ended
8:          mem = mem.apply_delayed_actions()
9:      end while
10:     return results
11: end function
```

VII. Automatic Synthesis of BTs

In this section, we present the third contribution of our paper. Having a BBT, we can trivially construct a corresponding BT because of each node type (Action, Condition, Control nodes) already has a BT definition. The execution of this BT will result in one of the physical states from the belief state. Therefore, we aim to construct first a BBT and then use corresponding BT for the runtime execution. For the inference we follow the definition in [4], which allows condition nodes to return running status. The planning pipeline was inherited from a previous work, that was aimed at BTs planning in a deterministic domain [5]. Below we highlight important differences before formally describe the planning algorithm. Briefly, the main routine step is still to find a failed condition and resolve it by inserting nodes or permuting branches (Alg.3). The planning is terminated when goal probability is achieved by means of self-simulation.

Algorithm 3 Planning routine

```
1:  function REFINE_TREE(initial_state, bt, goal_prob)
2:      bstate = BeliefMemory(state)
3:      prob = 0
4:      while prob < goal_prob do
5:          target = find_failed_condition(bt, bstate)   ▷ Section VII-B
6:          if threaten(target) then
7:              resolve_threat(bt, bstate, target)
8:          else
9:              resolve_by_insert(bt, bstate, target)   ▷ Section VII-C
10:         end if
11:         bstate = bt.simulate(initial_state)
12:         finished, bstate = bstate.split_by(r_i = 𝕊)
13:         prob = finished.probability()
14:     end while
15:     return results
16: end function
```

A. Input of the planning problem

In the previous work [5], a set of goal conditions is an input to the planning problem. If we want to set an action as a planning goal, we can define a goal as set of this action preconditions.

As we plan in a probabilistic domain, we should set a target success probability p_{goal} and terminate our planning algorithm as soon as constructed BT is expected to succeed with probability p_{goal}. In order to calculate it, as described in Section VI-D, we need to add an initial belief state for a planning problem. The current physical state of the robot could suite as an initial belief state for most applications.

B. Finding a condition to resolve

At this step, we need to choose the condition that is either not satisfied and there is no action in the BT that satisfies it or the condition whose value is unknown and there is no sensing action that makes such value known. In general, there could be many such conditions in BT. Consider for example the Fallback node in Fig. 4, in which the children are two actions with preconditions. If both actions have unsatisfied preconditions, we are unable to decide which action is preferred. In fact it might be that the one of the action's preconditions are easier to resolve: however this may be known only after further planning. Following previous on task planning with BT [5], we aim to find a *deepest failed* condition.

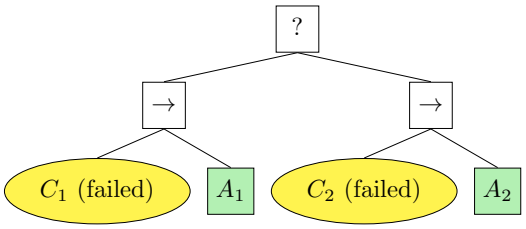


Fig. 4. An example of the ambiguous choice of the condition to resolve.

Assuming we ran a self-simulation procedure (Sec. VI-D), we end up in a belief state. Different scenarios of execution could infer different conditions to be "the deepest failed". Since we plan in belief space, for a BBT we have to choose the *most probable deepest failed* condition over those that do not hold in physical states. As we construct a tree in a way to avoid cases mentioned above, by "the deepest" we mean simply the deepest node in the tree. Hence, to find a condition to resolve C_k we need to find the deepest condition with the highest cumulative probability over all physical states:

$$k = argmax_i \sum_j p_j I_{i,j}$$

$I_{i,j} := 1$ if C_i is the deepest failed in physical state s_j, $I_{i,j} := 0$ otherwise.

C. Iterative Behavior Tree Expansion

There are two possible reasons why target precondition could hold other than $\mathbb{S}$ value. First, we check if there is a *conflict*, a postcondition of previously inserted action coinciding with the target precondition. Following the literature [5], we move tree branches to let this action be executed after the satisfaction of the target precondition. Then, if the precondition is still not satisfied or there was no conflict, we insert a *latched* action to either a skipper node or a fallback node, depending on the value that the condition holds (correspondingly, $\mathbb{R}$ or $\mathbb{F}$). The latched node is an node, that. Moreover, one can prevent reexecuting a whole subtree using a latch node [18]. At least one of the action postconditions has to result in a target condition satisfaction. In case of several possible actions, we can choose the action to insert by taking into account the following factors:

- the probability of a successful postcondition
- the fact if all action's preconditions are satisfied in a current belief state
- the history of previously executed actions

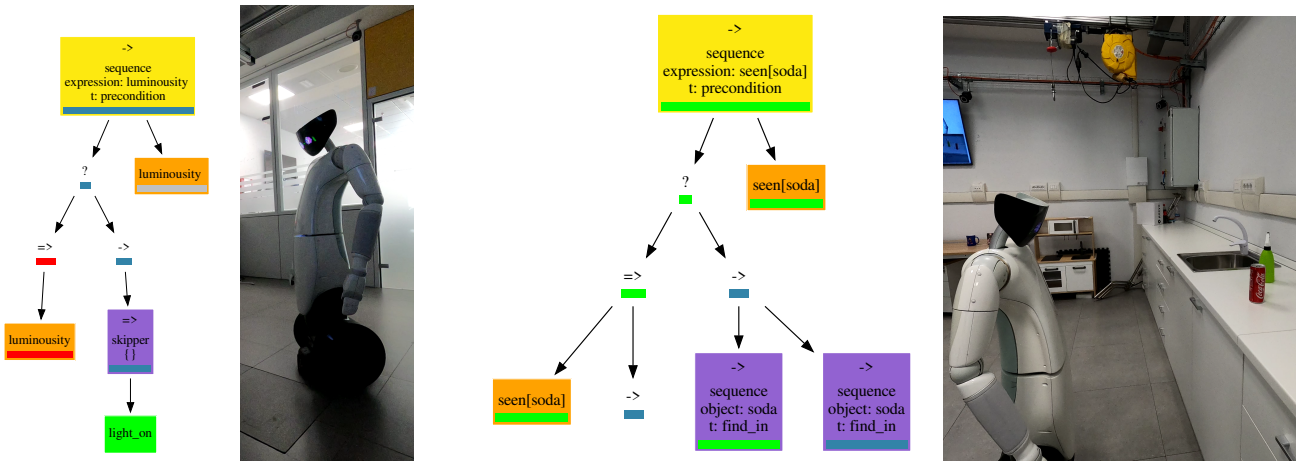


Fig. 5. In this experiment R1 was asked to look for the object $soda$. We hid the object from the $table2$ during first execution of the $find$ node template. We show two R1 states, each represented by a screenshot of the part of the BT visualization that is executed on the robot and a photo of the R1 at this moment. On th first pair of pictures R1 detected poor luminousity condition ($luminousity\_ok = \mathbb{F}$, condition node underscored with red on the screenshot) and commanded to turn on the light ($light\_on$). On the second pair of pictures we show an end of experiment, R1 is looking on the $soda$ can laying on the $table2$ place. Goal condition $seen(soda)$ is satisfied (returned $\mathbb{S}$ status, underscored with green on the screenshot). Running nodes are represented with a blue bar, nodes with success and failure statuses are with a green and red bar respectively.

Even though inserting *latched* actions reduces the reactivity of BT, we find this approach more flexible. We can insert an arbitrary number of latched actions to achieve the goal success probability. Inserted actions could be different in case there is more than one action that has a target postcondition. In addition, we can insert not just actions, but *node templates*, an arbitrary parameterized collection of nodes [19]. In the case when the result of a node template execution could be described simply by a set of postconditions, we can still assign postconditions to guide a planning algorithm. Any side effects shall be caught by self-simulation procedure and next steps of tree expansion.

VIII. Experimental Validation

In this section, we report the experimental validation. Framework implementation, specification of action and condition sets, and the example below are available online[4]. To validate our planning approach, we constructed a set of actions and conditions, that relates to the domestic application scenario. We used an off-the-shelf object detection pipeline to implement the condition $seen(object)$ that checks if an object is in the robot's camera field of view [20], [21]. We executed the generated BTs on the R1 robot [22] and achieved correct execution. For simplicity, in the paper we describe a subset of this domain and a simple goal. The domain is described by sets of actions (including node templates) (see Table II), conditions (see Table IIIa), and their parameter spaces (see Table IIIb).

The robot has to find a soda can with a probability of 0.9. The goal of this task is described by the condition $seen(soda)$ and goal probability was set to 0.9. The initial state of the robot was $seen(soda) = \mathbb{R}, at(table1) = \mathbb{F}, at(table2) = \mathbb{F}, luminousity\_ok = \mathbb{F}$ (omitting unused conditions). Let us follow the planner step by step. As the $seen(soda)$ was unobserved, the planner inserted the *perception* action $detect$ (see Fig. 6a). In order to make this observation, the condition $luminousity\_ok$ should hold $\mathbb{S}$. Therefore, $light\_on$ action was inserted (see Fig. 6b). After this step, the success probability is 0.5. As the target probability was set higher, and all failed states contained $seen(soda) = \mathbb{F}$ condition, we inserted a *find* sub-BT (see Fig. 6c). After this, success probability achieved 0.875. Note, that we did not need to insert extra $light\_on$ ac-

[4]https://github.com/safoex/pybt

Action	Params	Preconditions	Postconditions
goto	place	$\emptyset$	$0.95, at(place) = \mathbb{S}$ $0.05, no\ changes$
detect	object	$luminousity\_ok = \mathbb{S}$ $seen(object) = \mathbb{R}$	$0.5, seen(object) = \mathbb{S}$ $0.5, seen(object) = \mathbb{F}$
light_on		$\emptyset$	$1, luminousity\_ok = \mathbb{S}$
find	object	$seen(object) = \mathbb{F}$	$0.8, seen(object) = \mathbb{S}$ $0.2, seen(object) = \mathbb{F}$

TABLE II. Actions

Condition	Parameters	Values
at	place	$\mathbb{S}$, $\mathbb{F}$
seen	object	$\mathbb{S}$, $\mathbb{F}$, $\mathbb{R}$
luminousity_ok	$\emptyset$	$\mathbb{S}$, $\mathbb{F}$

(a) *Values* contains $\mathbb{R}$for conditions which might be unobservable.

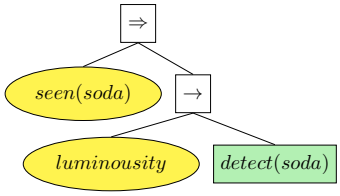
	Instances
place	table1, table2
object	soda, sprayer

(b) Parameters

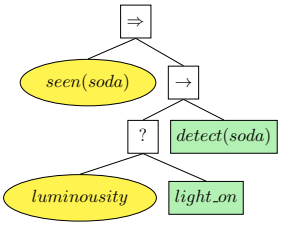
TABLE III. Conditions (a) and parameters (b), used in our validation scenario.

Nodes	Return statuses
condition	never started yet
action	running
template (expanded)	failed
template (not expanded)	success

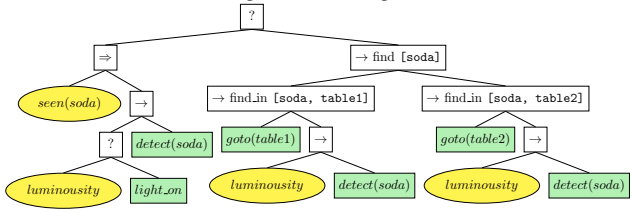
TABLE IV. Color legend for BT on Fig. 5



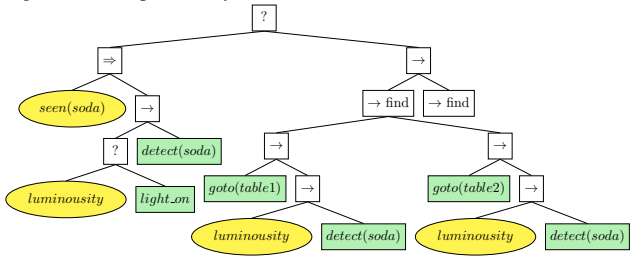
(a) We start from condition"seen(soda)". In case it is unobservable, we insert a Skipper node with "detect(soda)" action, which checks if soda is detected right now.



(b) It might happen that precondition "luminousity" of "detect" action could be unsatisfied. For this reason we insert action "light_on".



(c) In case we did not detect object "soda" on our current place, we search over possible locations. It is implemented by template "find" which consists of templates "find_in". Note, that we do not need to resolve condition"luminousity" anymore, as it must be already resolved by action "light_on" with probability 1 (check Table II.



(d) Final BT solution includes another attempt to find "soda" object. The second subtree "find" is not expanded on the picture to keep it compact.

Fig. 6. An example of iterative BT expansion step by step for the "look for soda" scenario on Fig. 5

tions because in all paths of execution, the precondition $luminousity\_ok = \mathbb{S}$ was already successfully satisfied. After the second $find$ template was inserted (not expanded on the Fig.6d), the success probability reached $\simeq 0.97$, and planning was terminated. Note, that setting the target probability closer to 1 would force the planner to insert more and more $find$ templates, pushing robot to make more and more attempts to find an object. Such behavior corresponds to one *not latched* node template $find$, which could be a result of planning in a not probabilistic domain.

IX. Conclusion

In this paper, we proposed a planning approach to automatically create BTs that takes into account state uncertainty. We extended the formulation of condition nodes, allowing them to represent the situation in which the value of a condition is unknown. This allowed us to handle both current and future state uncertainty. Our approach combines modularity and reactivity of BTs with automated planning. Planning in a probabilistic domain allowed us to control the target success probability. We demonstrated successful real robot execution of BT synthesized by our algorithm.

References

[1] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2009.

[2] S. Rabin, *Game AI Pro*, ch. 6. The Behavior Tree Starter Kit. CRC Press, 2014.

[3] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. Chapman and Hall/CRC Artificial Intelligence and Robotics Series, CRC Press, Taylor & Francis Group, 2018.

[4] E. Safronov, M. Vilzmann, D. Tsetserukou, and K. Kondak, "Asynchronous behavior trees with memory aimed at aerial vehicles with redundancy in flight controller," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3113–3118, Nov 2019.

[5] 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)*, pp. 8839–8845, IEEE, 2019.

[6] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1194–1227, 2013.

[7] M. Ghallab, D. Nau, and P. . Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[8] R. Platt Jr, R. Tedrake, L. Kaelbling, and T. Lozano-Perez, "Belief space planning assuming maximum likelihood observations," 2010.

[9] D. Hadfield-Menell, E. Groshev, R. Chitnis, and P. Abbeel, "Modular task and motion planning in belief space," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pp. 4991–4998, IEEE, 2015.

[10] B. Banerjee, "Autonomous acquisition of behavior trees for robot control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3460–3467, IEEE, 2018.

[11] S. Jones, M. Studley, S. Hauert, and A. Winfield, "Evolving behaviour trees for swarm robotics," in *Distributed Autonomous Robotic Systems*, pp. 487–501, Springer, 2018.

[12] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *IEEE Transactions on Games*, 2018.

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

[14] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE international conference on robotics and automation (ICRA)*, pp. 564–571, IEEE, 2017.

[15] K. French, S. Wu, T. Pan, Z. Zhou, and O. C. Jenkins, "Learning behavior trees from demonstration," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 7791–7797, IEEE, 2019.

[16] M. Colledanchise, R. M. Murray, and P. Ögren, "Synthesis of correct-by-construction behavior trees," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6039–6046, IEEE, 2017.

[17] A. Klökner, "Interfacing Behavior Trees with the World Using Description Logic," in *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

[18] A. Klöckner, "Behavior trees with stateful tasks," in *Advances in Aerospace Guidance, Navigation and Control*, pp. 509–519, Springer, 2015.

[19] E. Safronov, "Node templates to improve reusability and modularity of behavior trees," 2020.

[20] E. Maiettini, G. Pasquale, L. Rosasco, and L. Natale, "Speeding-up object detection training for robotics with falkon," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2018.

[21] E. Maiettini, G. Pasquale, L. Rosasco, and L. Natale, "On-line object detection: a robotics challenge," *Autonomous Robots*, Nov 2019.

[22] A. Parmiggiani, L. Fiorio, A. Scalzo, A. V. Sureshbabu, M. Randazzo, M. Maggiali, U. Pattacini, H. Lehmann, V. Tikhanoff, D. Domenichelli, A. Cardellino, P. Congiu, A. Pagnin, R. Cingolani, L. Natale, and G. Metta, "The design and validation of the r1 personal humanoid," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 674–680, Sep. 2017.