Towards Blended Reactive Planning and Acting using Behavior Trees

Michele Colledanchise, Diogo Almeida, and Petter Ögren

***Abstract*— In this paper, we study the problem of using a planning algorithm to automatically create and update a Behavior Tree (BT), controlling a robot in a dynamic environment. Exploiting the characteristic of BTs, in terms of modularity and reactivity, the robot continually acts and plans to achieve a given goal using a set of abstract actions and conditions.**

The construction of the BT is based on an extension of the Hybrid Backward-Forward algorithm (HBF) that allows us to refine the acting process by mapping the *descriptive* models onto *operational* models of actions, thus integrating the ability of planning in infinite state space of HBF with the continuous modular reactive action execution of BTs.

We believe that this might be a first step to address the recently raised open challenge in automated planning: the need of a hierarchical structure and a continuous online planning and acting framework.

We prove the convergence of the proposed approach as well as the absence of deadlocks and livelocks, and we illustrate our approach with examples from two different robotics domains.

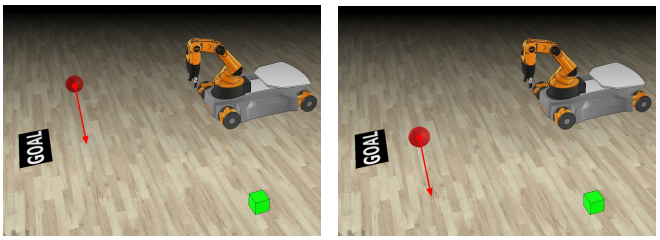
I. Introduction

To illustrate the need for blending reactive planning and acting, we consider the following simple example, depicted in Figure 1. A robot needs to plan and execute the actions needed to pick up an object, and place it in a given location. The environment is however dynamic and unpredictable. After pickup, the object might slip out of the robot gripper, or, as shown in Figure 1, external objects might move and block the path to the goal location.

In this paper, we propose an approach that produces reactive execution plans in the form of Behavior Trees (BTs). The BT includes reactivity, in the sense that if the object slips out of the robot gripper, it will automatically stop and pick it up again without the need to replan or change the BT. The BT also supports iterative plan refinement, in the sense that if an object moves to block the path, the original BT is extended to include a removal of the blocking obstacle. Then, if the obstacle is removed by an external actor, the BT reactively skips the obstacle removal, and goes on to pick up the main object without having to change the BT.

Within the AI community, there has been an increased interest in the combination of planning and acting, [1]–[3]. In particular, [1] describes two key open challenges:

- "Hierarchically organized deliberation. This principle goes beyond existing hierarchical planning techniques; its requirements and scope are significantly different. The actor performs its deliberation online"
- "Continual planning and deliberation. The actor monitors, refines, extends, updates, changes and repairs its plans throughout the acting process, using both descriptive and operational models of actions."



(a) (b)

Fig. 1. A simple example scenario where the goal is to place the green cube C onto the goal region G. The fact that the sphere S might be blocking the path must be handled. In (a) the nominal plan is *MoveTo(C)→Pick(C)→MoveTo(G)→Drop()* when the sphere suddenly blocks the path. In (b), after replanning, the plan is *MoveTo(S)→Push(S)→MoveTo(C) →Pick(C)→MoveTo(G)→Drop()* when the sphere moves away before being pushed.

Similarly, the recent book [2] describes the need for an agent that "reacts to events and extends, updates, and repairs its plan on the basis of its perception". Finally, [2] also notes that most of the current work in action planning yields a static plan, i.e., a sequence of actions that brings the system from the initial state to the goal state. Its execution is usually represented as a classical Finite State Machine (FSM). However, due to changes in the environment, the effect of an action can be unexpected. This may lead to situations where the agent replans from scratch on a regular basis, which can be expensive in terms of both time and computational load. We believe that BTs can provide an important step towards addressing these challenges.

BTs are a graphical mathematical model for reactive fault tolerant task executions. They were first introduced in the computer gaming industry [4] to control in game opponents, and is now an established tool appearing in textbooks [5], [6] and generic game-coding software such as Pygame[1], Craft AI [2], and the Unreal Engine[3]. BTs are appreciated for being highly modular, flexible and reusable, and have also been shown to generalize other successful control architectures such as the Subsumption architecture, Decision Trees [7] and the Teleo-reactive Paradigm [8]. So far, BTs are either created by human experts [9]–[14] or automatically designed

The authors are with the Robotics, Perception and Learning Lab, School of Computer Science and Communication, The Royal Institute of Technology - KTH, Stockholm, Sweden. e-mail: {miccol|diogoa|petter}@kth.se

[1] http://www.pygame.org/project-owyl-1004-.html
[2] http://www.craft.ai/
[3] https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/

using machine learning techniques [15]–[17] defining an objective function to maximize using heuristic methods.

In this paper we propose a formal automated mechanism to synthesize a BT. The construction of the tree is inspired by the Hybrid Backward-Forward algorithm (HBF), which was recently proposed as an action planner for dealing with infinite state spaces [18]. The HBF algorithm has been shown to efficiently solve problems with large state spaces. Using an HBF algorithm we can refine the acting process by mapping the *descriptive* model of actions, which describes *what* the actions do, onto an *operational* model, which defines *how* to perform an action in certain circumstances.

The proposed framework combines the planning capability in an infinite state space from HBF with the advantages of BTs compared to FSMs in terms of *reactivity* and *modularity*. Looking back at the example above, the *reactivity* of BTs enable the robot to pick up a dropped object without having to replan at all. The *modularity* enables extending the plan by adding actions for handling the blocking sphere, without having to replan the whole task. Finally, when the sphere moves away, once again the *reactivity* enables the correct execution without changing the plan. Thus, the proposed approach is indeed hierarchical and modular in its deliberation, and it does monitor, update and extend its plans while acting, addressing the needs described in [1], [2].

The main contribution of this paper is that we show how to use a planning algorithm (HBF) to iteratively create and refine BTs that represent reactive executions of the corresponding plans, based on abstract definitions of actions and conditions. To the best of our knowledge, this has not been done before.

The rest of this paper is organized as follows. In Section II we present related work, then in Section III we describe BTs and HBF. The problem formulation can be found in Section IV and the proposed approach in Section V. A theoretical analysis of the algorithm is given in Section VI. Some simulations are performed in Section VII to illustrate the approach, before concluding in Section VIII.

II. Related work

In this section we briefly summarize related work and compare it with the proposed approach. We focus on Automated Planning (AP) research as no work in the literature addressed our objective of automatically generating a BT from a set of abstract actions and conditions.

The AP community has developed solid solutions for solving path-finding in large state spaces. Such solution have found successful applications in a wide variety of problems. Nonetheless, numerous AP problems remain open challenges [1], [19]–[21]. For example, it was noted by Kaelbling et al. [22], that there is no systematic AP framework that can address an abstract goal such as "wash dishes" and reason on a long sequence of actions in dynamic or finite horizon environments.

In the robotic community most of the planners focus, without loss of generality, on manipulation planning where the objective is to have a robot operating in the environment with the possibility to modify, in a limited degree, the environment's configuration.

Early approaches treated the configuration space as continuous for both object and robot but used discrete actions [21], [23], [24], while our approach consider a descriptive model of actions and conditions (i.e. abstract templates).

Later work [25] proposed so-called *multi-modal planning*, as a generalization of manipulation planning, that is characterized by planning using different operational *modes* representing different constraint subspaces of the state space. These plans were characterized by switching between operating a single mode and choosing the mode. Multi-modal planning was then extended to address more complex problems combining a bidirectional search with an hierarchical strategy to determine the operational model of their actions [26]. However, in contrast with our approach, these works assume a static environment and disregard the importance of continually deliberating while acting.

Recent approaches to robotic planning combine discrete task planning and continuous motion planning frameworks [27]–[29] pre-sampling grasps and placements producing a family of possible high level plans. Despite their hierarchical architectures, those approaches do not consider the continual update of the current plan.

Hierarchical Planning in the Now (HPN) [3] works effectively in domains where planning in detail far into the future is intractable. It iteratively plans and executes actions to get closer to the goal, while avoiding to construct the whole plan in detail at once. While inspired by HPN, our approach differs from it mainly in the execution framework. The use of BTs allows the system to continually monitor the current plan and reactively preempt sub-plans when they are no longer needed due to an environmental change. Moreover if a sub-goal that was previously achieved is no longer achieved, our approach does not extend the current plan. Instead, it automatically invokes the sub-routine previously computed for that particular sub-goal.

Other approaches [30] consider two types of replanning: *aggressive replanning*, where replanning is done after every action; and *selective replanning*: where replanning is done whenever a new change in the environment is discovered or when a new path to the goal is found that is shorter than the existing plan by a given threshold. In our approach we replan only when needed. By using continual hierarchical monitoring, we are able to monitor the part of the environment that is relevant for goal satisfaction, disregarding environmental changes that do not affect our plan. This enables us to plan and act efficiently in highly dynamic environments.

Within the AI community, with a long history of focusing on state-space search methods, many have addressed the problem using heuristics that approximate the distance between the current and the goal configuration [31]. One approximation is characterized by the *delete relaxation* that assumes each sub-goal, once achieved, remains achieved for the duration of the plan [32]. This is too strict for planning in dynamic environments.

The work that is closest to this paper is [18] where a

Hybrid Backward Forward (HBF) algorithm was proposed as an action planner in infinite state space. HBF is a forward search in state space, starting at the initial state of the complete domain, repeatedly selecting a state that has been visited and an action that is applicable in that state, and computing the resulting state, until a state satisfying a set of goal constraints is reached. HBF was successfully used to efficiently solve the HPN. The advantage of this work lies in the restriction to the set of useful actions, building a so-called *reachability graph*. A backward search algorithm builds the reachability graph working backward from the goal's constraints, using them to drive sampling of actions that could result in states that satisfy them. However, even though HBF enables us to deal with infinite state space, the original formulation still derives a plan that is static. Hence it does not address the need of continually acting and planning in dynamic environments.

III. Background: BT and HBF

In this section we briefly describe BTs and HBF. A more detailed description of BTs can be found in [7], and one of HBF can be found in [18].

A. Behavior Trees

BTs are a graphical modeling language and a representation for execution of actions based on conditions and observations in a system.

A BT is a directed rooted tree where each node is either a control flow node or an execution node. For each connected node we define as *parent* the outgoing node and *child* the incoming node. The root is the single node without parents, whereas all other nodes have one parent. The control flow nodes have one or more children, and the execution nodes have no children. Graphically, the children of nodes are placed below it. The children nodes are executed in the order from left to right, as shown in Figures 2-3.

The execution of a BT begins from the root node. It sends *ticks* [4] with a given frequency to its child. When a parent sends a tick to a child, the execution of this is allowed. The child returns to the parent a status *running* if its execution has not finished yet, *success* if it has achieved its goal, or *failure* otherwise.

There are four types of control flow nodes (fallback, sequence, parallel, and decorator) and two execution nodes (action and condition). Below we describe the execution of the nodes used in this paper.

Fallback: The fallback[5] node ticks its children from the left, returning success (running) as soon as it finds a child that returns success (running). It returns failure only if all the children return failure. When a child returns running or success, the fallback node does not tick the next child (if any). The fallback node is graphically represented by a box with a "?", as in Figure 2.

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

[5]Fallbacks are sometimes also called Selectors

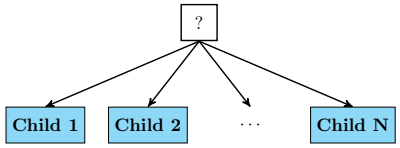


Fig. 2. Graphical representation of a fallback node with N children.

Sequence: The sequence node ticks its children from the left, returning 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 return running or failure, the sequence node does not tick the next child (if any). The sequence node is graphically represented by a box with a "→", as in Figure 3.

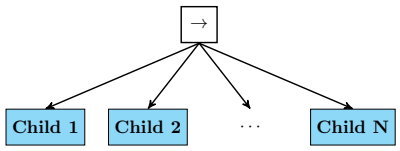


Fig. 3. Graphical representation of a sequence node with N children.

(a) (b)

Fig. 4. Graphical representation of (a) an action and (b) a condition node.

Action: The action node performs an action, returning success if the action is completed and failure if the action cannot be completed. Otherwise it returns running. An action node is shown in Figure 4(a)

Condition: The condition node checks if a condition is satisfied or not, returning success or failure accordingly. The condition node never returns running. A condition node is shown in Figure 4(b)

To get familiar with BT notation, and prepare for the coming sections, we look at a BT plan addressing the simple example in Section I. The BT was created using the proposed approach as will be explained in Section V, but for now we just focus on how it is executed.

Example 1: The robot in Figure 1 is given the task to move the green cube into the rectangle marked GOAL. Ignoring the presence of the red sphere, a reactive plan BT can be found in Figure 5(e). Each time step, the root of the BT is ticked. The root is a fallback which ticks is first child, the condition $o_c \in GoalRect$ (cube on goal). If the cube is indeed in the rectangle we are done, and the BT returns Success.

If not, the second child, a sequence, is ticked. The sequence ticks its first child, which is a fallback, which again ticks its first child, the condition $h = c$ (object in hand is cube). If the cube is indeed in the hand, the condition returns success, its parent, the fallback returns success, and its parent, the sequence ticks its second child, which is a different fallback, ticking its first child which is the condition $o_r \in \mathcal{N}_{p_g}$ (robot in the neighborhood of p_g). If the robot is in the neighborhood of the goal, the condition and its fallback parent returns success, followed by the sequence ticking its

third child, the action $Place(c, p_g)$ (place cube in a position p_g on the goal), and we are done.

If $o_r \in \mathcal{N}_{p_g}$ does not hold, the action $MoveTo(p_g, \tau_g)$ (move to position p_g using the trajectory τ_g) is executed, given that the trajectory is free $\tau \subset C_{ollFree}$. Similarly, if the cube is not in the hand, the robot does a $MoveTo$ followed by a $Pick(c)$ after checking that the hand is empty, the robot is not in the neighborhood of c and that the corresponding trajectory is free.

We conclude the example by noting that the BT is ticked every timestep, e.g. every 0.1 second. Thus, when actions return running (i.e. they are not finished yet) the return status of running is progressed up the BT and the corresponding action is allowed to control the robot. However, if e.g., the cube slips out of the gripper, the condition $h = c$ instantly returns failure, and the robot starts checking if it is in the neighborhood of the cube or if it has to move before picking it up again.

B. The Hybrid Backward Forward algorithm

HBF was recently proposed in [18] as an AP for infinite state space problems. Given an initial state $s_0 \in \mathcal{S}$, a set of action templates, and a set of goal constraints $\Gamma \subset \mathcal{S}$ (we use Γ to denote both the set of constraints, and the set of states that satisfy the constraints), the plan is a sequence $a_1, a_2, \ldots, a_m$ of actions such that $a_m.\text{eff}(a_{m-1}.\text{eff}(\ldots a_1.\text{eff}(s_0))) \in \Gamma$ where $a_i.\text{eff}(s_j)$ is the effect of action a_i performed in state s_j. This plan is static. It assumes that $a_i.\text{eff}(s_i) \in a_{i+1}.\text{con}$, where $a_{i+1}.\text{con}$ is the set of conditions that needs to be satisfied to successfully perform the action a_{i+1}. It uses backward search in a subspace of the problem space to efficiently create a Reachability Graph (RG). The creation of the RG starts from the goal constraint and finds a combination of valid samples for the actions templates. These valid samples are then used to identify *executable* actions that are not in conflict with each other (e.g. If the robot has to drop an object A to be able to pick an object B, it will not drop the object A on top of B). For a detailed description of HBF we refer to [18].

IV. Problem Formulation

In this section we first make a set of assumptions, then state the main problem.

Assumption 1: The agent can verify if an action has succeeded, failed or if it is running.

Assumption 2: The agent can verify if a condition is true or false.

Assumption 3: For each initial state there exists a sequence of actions that lead the system to the given goal.

Assumption 4: The effect of the dynamic environment can void the satisfaction of sub-goals at most a finite number of times.

Assumption 5: Given two actions a_i and a_j, if the execution of a_i requires the execution of a_j, a_j must not require the execution of a_i.

Assumption 6: All actions are ultimately reversible. That is, each action can be undone trough a finite sequence of actions.

Algorithm 1: main loop

```
1  T ← ∅
2  for c in C_goal do
3      T ← Sequence(T, c)
4  while True do
5      T ← RefineActions(T)
6      do
7          r ← Execute(T)
8      while r ≠ FAILURE
9      c_f ← GetConditionToExpand(T)
10     T, T_new_subtree ← ExpandTree(T, c_f)
11     while Conflict(T) do
12         T ← IncreasePriority(T_new_subtree)
```

Definition 1: $\mathcal{S} \triangleq \mathbb{R}^n$ is the system state space. It captures all relevant aspects of the overall system.

Definition 2: $s \in \mathcal{S}$ is the system state. It is an element of the system state space. Each component s_i is a system state variable.

Definition 3: $c_i : \mathcal{S} \to \{0, 1\}$ is a simple constraint and $\mathcal{C} = \{c_i\}$ is the set of all constraints.

Definition 4: A state $s \in \mathcal{S}$ is said to *support* a constraint $c \in \mathcal{C}$ (written $s \vdash c$) if and only if $c(s) = 1$.

Definition 5: α is an action template. It contains the descriptive model of an action. An action template is characterized by conditions $\alpha_{con} \subset \mathcal{C}$ and effects $\alpha_{eff} \subset \mathcal{C}$. Conditions and effect are both constraints.

Definition 6: $a : \mathcal{S} \to \mathcal{S}$ is an action primitive. It contains the operational model of an action and is executable.
Assigning a value to each variable in an action template yields an action primitive.

Assumption 7: For each action template there exists at least one corresponding action primitive.

Problem 1: Given a set of goal constraints $\mathcal{C}_{goal} \subset \mathcal{C}$ and a set of action templates, derive a BT that leads the agent to a state s_{final} such that $s_{final} \vdash c, \forall c \in \mathcal{C}_{goal}$ independently of the starting state.

V. Proposed Approach

Formally, the proposed approach is described in Algorithms 1 and 2. First we will give an overview of the algorithms and see how they are applied to the problem described in *Example 1*, to iteratively create the BTs of Figure 5. We will then discuss the key steps in more detail.

A. Algorithm Overview

Running Algorithm 1 we have the set of goal constraint $\mathcal{C}_{goal} = \{o_c \in \{\text{GoalRect}\}\}$, thus the initial BT is composed of a single condition $\mathcal{T} = (o_c \in \{\text{GoalRect}\})$, as shown in Figure 5(a). The first iteration of the loop starting on Line 4 of Algorithm 1 now produces the next BT shown in Figure 5(b), and the second iteration produces the BT in Figure 5(c) and so on until the final BT in Figure 5(e).

Algorithm 2: Behavior Tree Expansion

```
1  Function ExpandTree(T, c_f)
2      A_T ← GetAllActTemplatesFor(c_f)
3      T_fall ← c_f
4      for a in A_T do
5          T_seq ← ∅
6          for c_a in a.con do
7              T_seq ← Sequence(T_seq, c_a)
8          T_seq ← Sequence(T_seq, a)
9          T_fall ← Fallback(T_fall, T_seq)
10     T ← Substitute(T, c_f, T_fall)
11     return T, T_fall
```

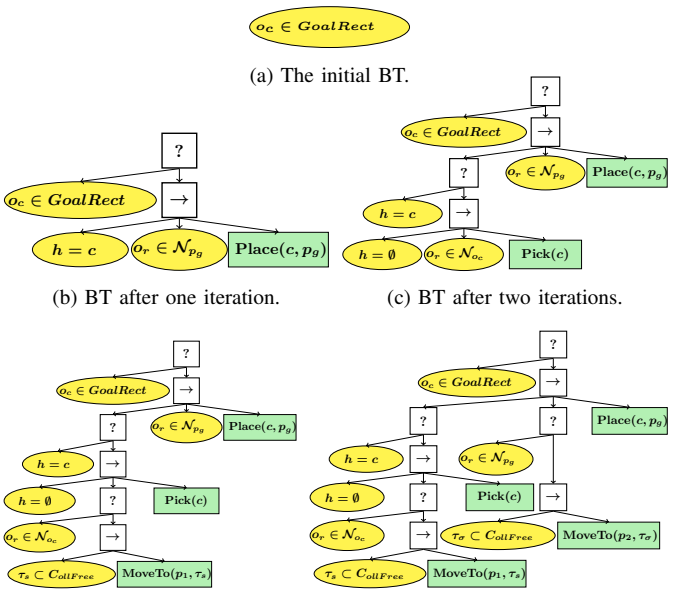
Algorithm 3: Get Condition to Expand

```
1  Function GetConditionToExpand(T)
2      for c_next in GetConditionsBFS() do
3          if c_next.status = FAILURE and
             c_next ∉ ExpandedNodes then
4              ExpandedNodes.push_back(c_next)
               return c_next
5      return None
```

In detail, running $\mathcal{T}$ on Line 7 returns a failure, since the cube is not in the goal area. Trivially, the *GetCondition-ToExpand* returns $c_f = (o_c \in \{\text{GoalRect}\})$, and a call to ExpandTree (Algorithm 2) is made on Line 10. On Line 2 of Algorithm 2 we get all action templates that satisfy c_f i.e. $A_T = Place$. Then on Line 7 and 8 a sequence $\mathcal{T}_{seq}$ is created of the conditions of $Place$ (the hand holding the cube $h = c$ and the robot being near the goal area $o_r \in \mathcal{N}_{p_g}$) and $Place$ itself. On Line 9 a fallback $\mathcal{T}_{seq}$ is created of c_f and the sequence above. Finally, a BT is returned where this new sub-BT is replacing c_f. The resulting BT is shown in Figure 5(b).

Note that Algorithm 2 describes *the core principle* of the proposed approach. *A condition is replaced by a check if the condition is met, and an action to meet it that is only executed if needed.* If there are several such actions, these are added as fallbacks. Finally, the action is preceded by conditions checking its own preconditions. If needed, these conditions will be expanded in the same way in the next iteration.

Running the next iteration of Algorithm 1, a similar expansion of the condition $h = c$ transforms the BT in Figure 5(b) to the BT in Fig. 5(c). Then, an expansion of the condition $o_r \in \mathcal{N}_{o_c}$ transforms the BT in Figure 5(c) to the BT in Figure 5(d). Finally, an expansion of the condition $o_r \in \mathcal{N}_{p_g}$ transforms the BT in Figure 5(d) to the BT in Figure 5(e), and this BT is able to solve *Example 1*. Figure 6 shows the resulting RG used for action refinements (Algorithm 1 Line 5), see [18] for details.



(a) The initial BT.

(b) BT after one iteration.

(c) BT after two iterations.

(d) BT after three iterations.

(e) BT after four iterations. Final Tree

Fig. 5. BT updates during the execution.

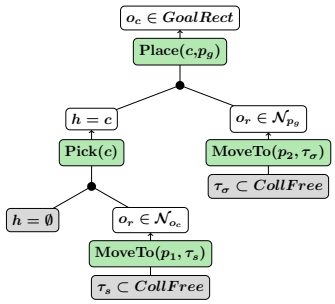


Fig. 6. Reachability Graph used to find the action refinement for the final tree of Figure 5(e).

B. The Algorithm Steps in Detail

1) Refine Actions (Algorithm 1 Line 5): To plan in infinite state space we rely on the Reachability Graph (RG) provided by the HBF algorithm [18]. The RG provides efficient sampling for the actions in the BT, allowing us to map the descriptive model of an action into its operational model. The samples provided by the RG are not in conflict with each other, and the process implements the action refinement described in [20].

2) Get Condition To Expand and Expand Tree (Algorithm 1 Lines 9 and 10) : If the BT returns failure, Line 9 invokes Algorithm 3, that finds the condition to expand by searching through the conditions returning failure in a Breadth First Search (BFS) fashion. If no such condition is found (Algorithm 3 Line 5) that means that an action returned failure due to an old refinement that is no longer valid. In that case, at the next loop of Algorithm 1 a new refinement is found (Algorithm 1 Line 5) and under Assumption 7 such a refinement always exists. If Algorithm 3 returns a condition, this will be expanded (Algorithm 1 Line 10), as shown in the example of Figure 5. Example 3 below highlights the BFS expansion. Thus, $\mathcal{T}$ is expanded until

it can perform an action (i.e. until $\mathcal{T}$ contains an action template whose condition are supported by the initial state). In Section VI we will prove that $\mathcal{T}$ is expanded a finite number of times. If there exists more than one valid action that satisfies a condition, their respective trees (sequence composition of the action and its conditions) are collected in a fallback composition, which implements the different options the agent has to satisfy such a condition. Note that at this stage we do not investigate which action is the optimal one. As stressed in [2] the cost of minor mistakes (e.g. non optimal actions execution) is often much lower than the cost of the extensive modeling, information gathering and thorough deliberation needed to achieve optimality.

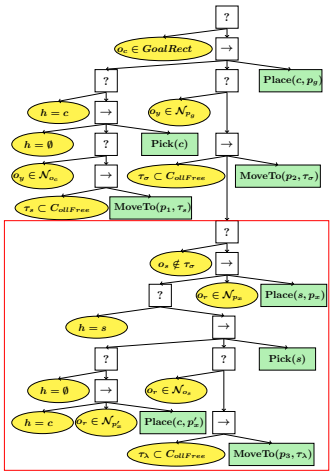
3) Conflicts and Increases in Priority (Algorithm 1 Lines 11 and 12): Similar to any STRIPS-style planner, adding a new action in the plan can cause a *conflict* (i.e. the execution of this new action creates a mismatch between effects and preconditions in the progress of the plan). In our framework, this situation is checked in Algorithm 1 Line 11 by analyzing the conditions of the new action added with the effects of the actions that the subtree executes before executing the new action. If this effects/conditions pair is in conflict, the goal will not be reached. An example of this situation is described in Example 2 below.

Again, following the approach used in STRIPS-style planners, we resolve this conflict by finding the correct action order. Exploiting the structure of BTs we can do so by moving the tree composed by the new action and its condition leftward (a BT executes its children from left to right, thus moving a sub-tree leftward implies executing the new action earlier). If it is the leftmost one, is means that it must be executed before its parent (i.e. it must be placed at the same depth of the parent but to its left). This operation is done in Algorithm 1 Line 12. We incrementally increase the priority of this subtree in this way, until we find a feasible tree. In Section VI we prove that a feasible tree always exists.

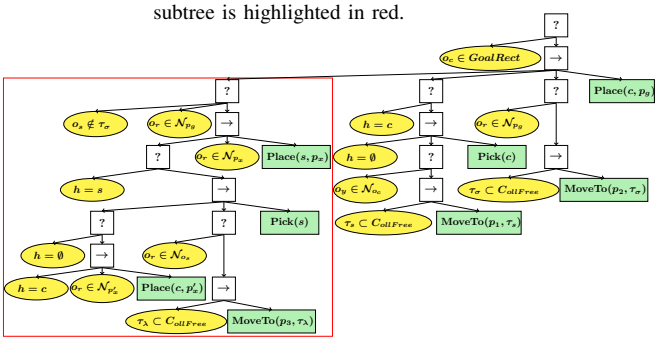
Example 2: Here we show a more complex example highlighting two main properties of our approach: the livelock freedom and the continual deliberative plan and act cycle. This example is an extension of Example 1 where, due to the dynamic environment, the robot needs to replan.

Consider the execution of the final BT in Figure 5(e) of Example 1, where the robot is carrying the desired object to the goal location. Suddenly, as in Figure 1 (b), an object s obstructs the (only possible) path. Then the condition $\tau \subset C_{ollFree}$ returns failure and Algorithm 1 expands the tree accordingly (Line 10) as in Figure 7(a).

The new subtree has as condition $h = \emptyset$ (no objects in hand) but the effect of the left branch (i.e. the main part in Figure 5(e)) of the BT is $h = c$ (cube in hand) (i.e. the new subtree will be executed if and only if $h = c$ holds). Clearly the expanded tree has a conflict (Algorithm 1 Line 11) and the priority of the new subtree is increased (Line 12), until the expanded tree is in form of Figure 7(b). Now the BT is free from conflicts as the first subtree has as effect $h = \emptyset$ and the second subtree has a condition $h = \emptyset$. Executing the tree the robot approaches the obstructing object, now



(a) Unfeasible expanded tree. The new subtree is highlighted in red.



(b) Expanded Feasible subtree.

Fig. 7. Steps to increase the priority of the new subtree added in Example 2.

the condition $h = \emptyset$ returns failure and the tree is expanded accordingly, letting the robot drop the current object grasped, satisfying $h = \emptyset$, then it picks up the obstructing object and places it outside the path. Now the condition $\tau \subset C_{ollFree}$ finally returns success. The robot can then again approach the desired object and move to the goal region and place the object in it.

4) Get All Action Templates: Let's look again at *Example 1* and see how the BT in Figure 5(e) was created using the proposed approach.

In this example, the action templates are summarized below with conditions and effect:

MoveTo(p, τ)	Pick(i)	Place(i, p)
con : $\tau \subset C_{ollFree}$	con : $o_r \in \mathcal{N}_{o_i}$ $h = \emptyset$	con : $o_r \in \mathcal{N}_p$ $h = i$
eff : $o_r = p$	eff : $h = i$	eff : $o_i = p$

where τ is a trajectory, $C_{ollFree}$ is the set of all collision free trajectories, o_r is the robot pose, p is a pose in the state space, h is the object currently in the end effector, i is the label of the i-th object in the scene, and $\mathcal{N}_x$ is the set of all the poses near the pose x.

The descriptive model of the action *MoveTo* is parametrized over the destination p and the trajectory τ. It

requires that the trajectory is collision free ($\tau \subset C_{ollFree}$). As effect the action *MoveTo* places the robot at p (i.e. $o_r = p$); The descriptive model of the action *Pick* is parametrized over object i. It requires having the end effector free (i.e. $h = \emptyset$) and the robot to be in a neighbourhood $\mathcal{N}_{o_i}$ of the object i. (i.e. $o_r \in \mathcal{N}_{o_i}$). As effect the action Pick sets the object in the end effector to i (i.e $h = i$); Finally, the descriptive model of the action *Place* is parametrized over object i and final position p. It requires the robot to hold i, (i.e. $h = i$), and the robot to be in the neighbourhood of the final position p. As effect the action *Place* places the object i at p (i.e. $o_i = p$).

C. Comments on Algorithm

It is clear that this type of continual plan and act exhibits both the important principles of deliberation stressed in [1], [2]: Hierarchically organized deliberation and Continual online deliberation. For example, if the robot drops the object then the condition $h = c$ is no longer satisfied and the BT will execute the according subtree to pick the object. With no need for re-planning. This type of deliberative reactiveness is built into BTs. On the other hand, if during its navigation a new object pops up obstructing the robot's path, the condition $\tau \subset C_{ollFree}$ will no longer return success and the BT will be expanded accordingly. This case was described in Example 2. Moreover, note that we refine every time the tree returns failure. This is to encompass the case where an older refinement is no longer valid. Is such cases an action will return failure. This failure is propagated up to the root. The function *ExpandTree* (Algorithm 1 Line 10) will return the very same tree (the tree needs no extension as there is no failed condition of an action) which gets re-refined in the next loop (Algorithm 1 Line 5). For example, if the robot planned to place the object onto a precise position of the desk but then this position is no longer feasible (e.g. another object was placed in that position meanwhile).

D. Algorithm Execution on Graphs

Here, for illustrative purposes, we show the result of our approach when applied to a standard shortest path problem in a graph.

Example 3: Consider an agent moving in different states modeled by the graph in Figure 8 where the initial state is s_0 and the goal state is s_g. Every arc represents an action that moves an agent from one state to another. The action that moves the agent from a state s_i to a state s_j is denoted by $s_i \rightarrow s_j$. The initial tree, depicted in Figure 9(a), is defined as a condition node s_g which returns success if and only if the robot is at the state s_g in the graph. The current state is s_0 (the initial state). Hence the BT returns a status of *failure*. Algorithm 1 invokes the BT expansion routine. The state s_g can be reached from the state s_5, through the action $s_5 \rightarrow s_g$, or from the state s_3, through the action $s_3 \rightarrow s_g$. The tree is expanded accordingly as depicted in Figure 9(b). Now executing this tree, it returns a status of *failure*. Since the current state is neither s_g nor s_3 nor s_5. Now the tree is expanded in a BFS fashion, finding a sub-tree for condition s_5 as in Figure 9(c). The process continues for two more iterations. Note that at iteration 4 (See Figure 10(b)) Algorithm 1 did not expand the condition s_g as it was previously expanded (Algorithm 3 line 3) this avoids infinite loops in the search. The same argument applies for conditions s_4 and s_g in iteration 5 (See Figure 10(c)). The BT at iteration 5 includes the action $s_0 \rightarrow s_1$ whose precondition is satisfied (the current state is s_0). The action is then executed. Performing that action (and moving to s_1), the condition s_1 is satisfied. The BT executes the action $s_1 \rightarrow s_3$ and then $s_3 \rightarrow s_g$, making the agent reach the goal state.

It is clear that the resulting execution is the same as a BFS on the graph would have rendered. Note however that the algorithm is designed for more complex problems than graph search.

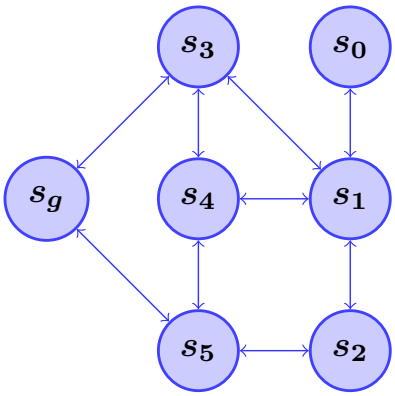
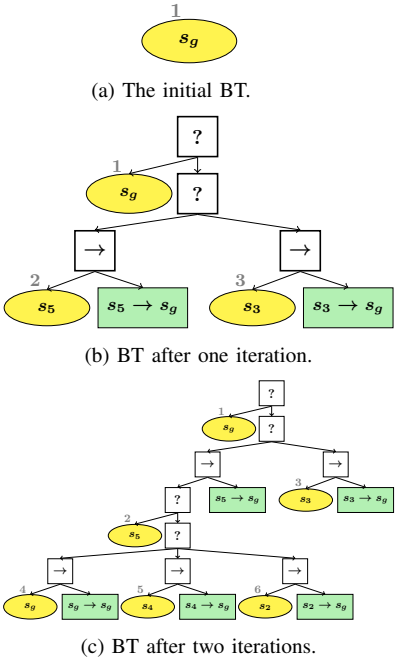


Fig. 8. Graph representing the task of Example 3.

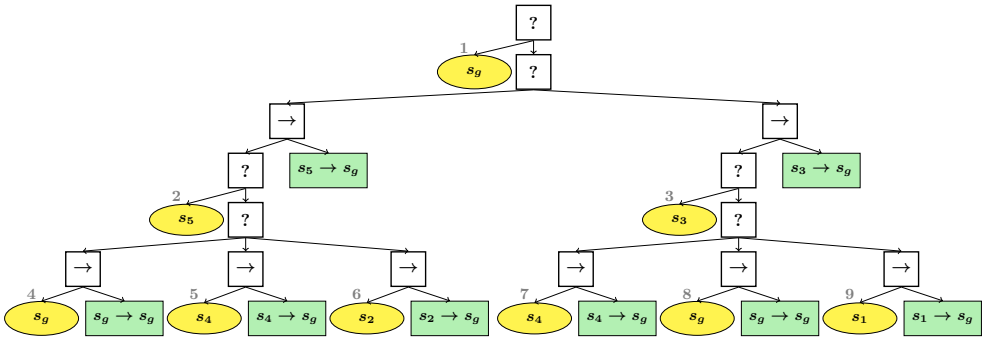


(a) The initial BT.

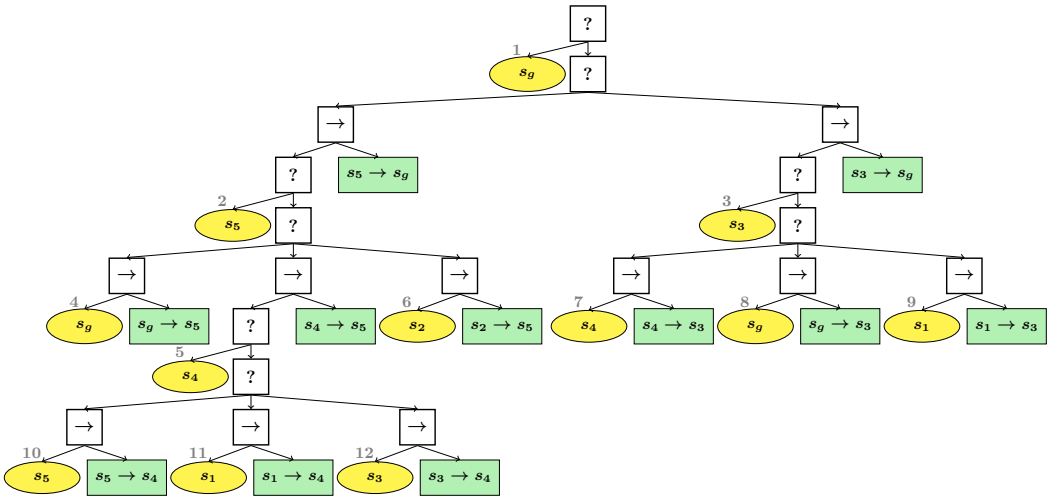
(b) BT after one iteration.

(c) BT after two iterations.

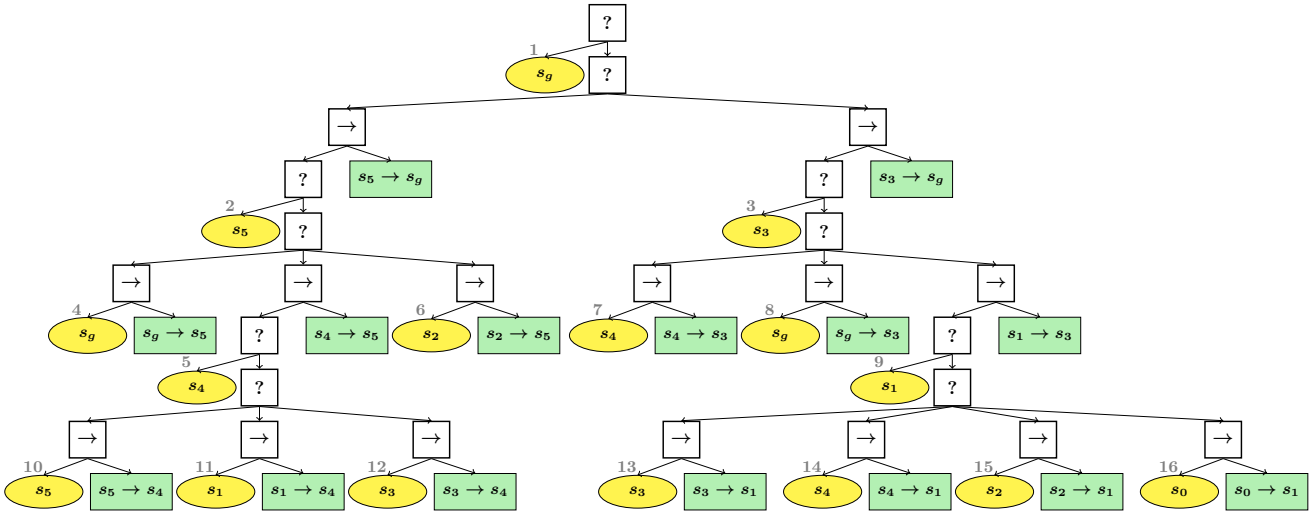
Fig. 9. First three BT updates during execution of Example 3. The numbers represent the index of the BFS of Algorithm 3.



(a) BT after three iterations.



(b) BT after four iterations.



(c) BT after five iterations.

Fig. 10. Next BT updates during execution of Example 3. The numbers represent the index of the BFS of Algorithm 3.

VI. Theoretical Analysis

In this section we discuss the framework from a theoretical standpoint. Since we are blending planning and acting, standard notions of correctness and completeness from planning do not hold. As in [3], our correctness criterion is that if there exist a finite sequence of operations such that the goal state is reachable from the initial state, then Algorithm 1 will execute such a sequence of operations.

In a discrete domain and without any hierarchical refinements, a regression planner under Assumption 3 would produce a correct plan. (See Example 3). The authors of [3] prove that the refinements do not affect the ability to achieve the global goal.

The correct plan of a regression planner would be executed action by action in a FSM fashion. However, the execution framework of our approach is represented by a BT. Thus we need to examine the effect of using BTs as the task execution framework. If the global goal and the preconditions for each action were described by, at most, a single condition, there would be no choice of which condition to satisfy first, and the execution of the BT would be a reactive version of the one of a FSM of a regression planner (reactive in the sense that it continually monitors the plan and executes the correct sub-tree). When an action, or the global goal, has multiple preconditions, the order in which those are satisfied must be defined. It is often the case that the satisfaction of a precondition can void the satisfaction of another precondition previously satisfied. Since BTs continually monitor the satisfaction of preconditions, and re-execute the subtree if a precondition is no longer satisfied, this could lead to a livelock (i.e. a loop in the actions to satisfy the preconditions). Thus, we need to prove that our approach is livelock free.

Lemma 1: A conflict free BT always exists under Assumptions 3, 5 and 6.

Proof: Whenever the expansion of the BT adds a new action template α, this may cause a conflict. If α causes a conflict, its refinement must be executed before another action primitive. Since this other action is not identified yet, the function *increase priority* (Algorithm 1 Line 12) manipulates the order in which α is executed. Under Assumptions 5 and 3 a correct order for that action exists. Under Assumption 6 the actions previously executed that caused the conflict can be undone by a finite sequence of actions. ■

Lemma 2: The proposed approach is livelock free.

Proof: To prove this lemma we must prove that our approach does not execute infinite loops in the system state space from s_{curr} to any $s \in \mathcal{S}$ reachable from s_{curr}. Under Assumption 3, there exists a sequence of actions that moves the system state from s_{curr} to s_{final}. From Lemma 1 each action in the BT executed does not jeopardize the execution of the others, that is the satisfaction of a sub-goal (e.g. a precondition of an action) does not void the satisfaction of another sub-goal. Under Assumption 4 the environment does not force the agent to execute infinite loops of actions by keeping nullifying sub-goals. ■

Before proving that Algorithm 1 solves Problem 1 we need to prove that for each sub-goal, as well as for the global goal, Algorithm 1 finds the action to perform in finite time.

Lemma 3: For each goal, Algorithm 1 finds the action to perform in finite time.

Proof: The search is done backward from the goal to a constraint that is satisfied by the current state. Under Assumption 3 the backward search finds a finite sequence of actions. ■

Proposition 1: Algorithm 1 solves Problem 1.

Proof: If the current state s is such that $s \vdash c_{goal}$, then the goal is satisfied and the BT returns success, executing no action. If $s \nvdash c_{goal}$, then two situation may occur: T (the refined BT) returns running. This means the tree is executing an action to satisfy a sub-goal; T returns failure. This means that either a new action refinement is needed (which always exists under Assumption 7) or a new plan refinement is needed and $\mathcal{T}$ (the template BT) must be expanded to satisfy a sub-goal (or the global goal). Lemma 3 proves that this expansion terminates in finite time. Each sub-goal is satisfied finding a sequence of action in a BFS fashion and Lemma 2 shows that the satisfaction of a sub-goal does not void the satisfaction of another sub-goal.

■

VII. Results

In this section we show how the proposed approach scales to complex problems using two different scenarios. First, a KUKA Youbot scenario, where we show the applicability of our approach on dynamic and unpredictable environments, highlighting the importance of continually planing and acting. Second, an ABB Yumi industrial manipulator scenario, where we highlight the applicability of our approach to real world plans that require the execution of a long sequence of actions. The experiments were carried out using the physics simulator V-REP, in-house implementations of low level controllers for actions and conditions and an open source BT library[6]. Figures 11 and 12 show the execution of two KUKA youbot experiments and Figure 13 show the execution of one ABB Yumi robot experiment. A video showing the executions of all the experiments is publicly available[7].

A. KUKA Youbot experiments

In these scenarios, which are an extension of Examples 1 and 2, a KUKA Youbot has to place a green cube on a goal area, see Figures 11 and 12. The robot is equipped with a single arm with a simple parallel gripper. Additional objects may obstruct the feasible paths to the goal, and the robot has to plan when to pick and where to place to the obstructing objects. Moreover external actors may co-exist in the scene and force the robot to replan by modifying the environment (e.g. picking and placing objects around).

[6] http://wiki.ros.org/behavior_tree

[7] https://youtu.be/b_Ug2My9_Xw

B. ABB Yumi experiments

In these scenarios, an ABB Yumi has to assemble a cellphone, whose parts are scattered across a table, see Figure 13. The robot is equipped with two arms with simple parallel grippers, which are not suitable dexterous manipulation. Some parts must be grasped in a particular position. For example the opening on the cellphone's chassis needs to face away from the robot's arm, exposing it for the assembly. However, the initial position of a part can be such that it requires multiple grasps transferring the part to the other gripper, effectively changing its orientation w.r.t the grasping gripper. See video link above.

VIII. Conclusions

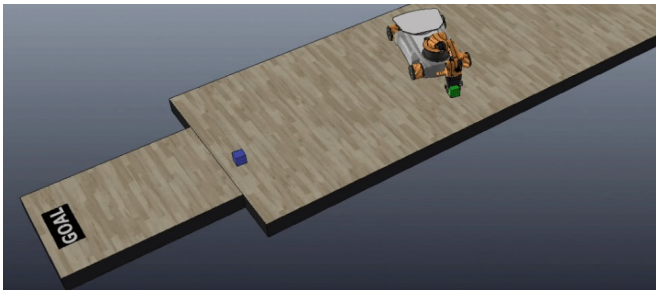
In this paper we proposed a framework to automatically synthesize a BT that satisfies a goal given a set of abstract action and condition nodes, to the best of our knowledge, this has not been done before. The framework combines the advantages of BTs, in terms of modularity and reactivity, with the infinite dimensional planning capability of the HBF algorithm. The resulting approach is an attempt to address the challenges regarding blending planning and acting that have been identified in the planning community. We analyzed the framework from a theoretical standpoint and showed its applicability in dynamic and unpredictable scenarios.

IX. Future Work

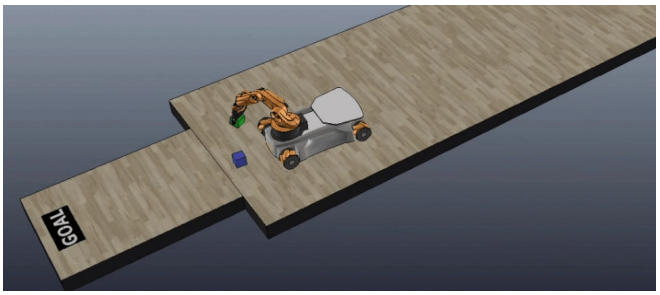
We will investigate the integration of this type of blended planning and acting with the learning of action templates. In particular we are interested in learning the preconditions and effects for each action useful for the planning. Recent work [17] shows how BTs are suitable for reinforcement learning. Note however that the challenges we will face are slithgly different that the one addressed in [17]. We are not able to define an objective functions that is maximized when the preconditions are found. Rougly speaking [17], finds a *sufficient* set of preconditions of an action for a *given* task. We will be interested in finding *necessary* and *sufficient* set of preconditions of an action for *any* task. We believe that the modularity of BTs will allow us to learn individual pieces for preconditions and effects to characterize the action templates. The learning process could be supervised by a domain expert, taking inspirations from [33], [34] or carryied out by interacting with the environment, taking inspirations from [35]. The learning process can be integrated in a sequential fashion (i.e. the approach learns first the action templates and then blends planning and acting.) or in a interleaved fashion (the learning of preconditions and effects gets blended with planning and acting.). We believe that the particular structure and execution of BTs effectively allows blending learning, acting and planning.

X. Acknowledgments

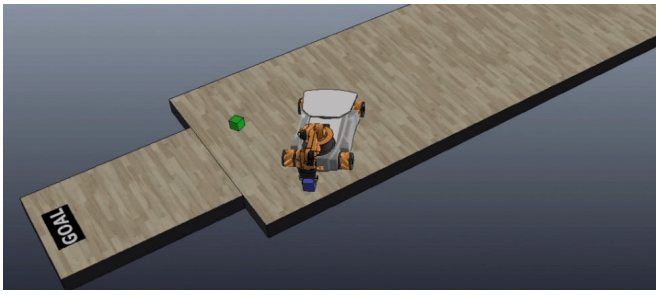
This work has been supported by the SARAFun project, partially funded by the EU within H2020 (H2020-ICT-2014/H2020-ICT-2014-1) under grant agreement no. 644938. The authors gratefully acknowledge the support.



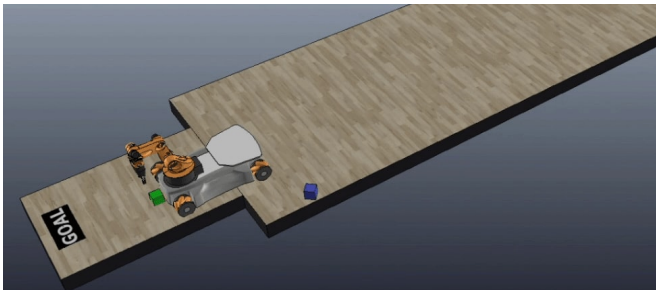
(a) The robot picks the desired object: a green cube.



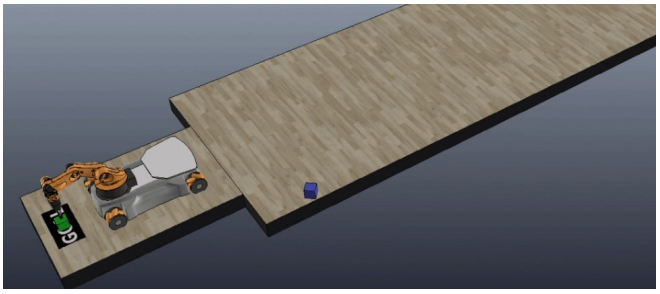
(b) The robot has to place the blue cube away from the path to the goal. But the robot is currently grasping the green cube. Hence the sub-tree created to place the blue cube away needs to have a higher priority.



(c) The blue cube is placed at the side.

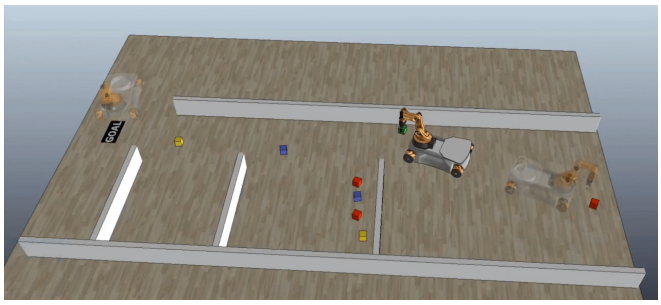


(d) While the robot is reaching the goal region, the green cube slips out of the gripper. The robot reactively preempts the sub-tree to move to the goal and re-executes the sub-tree to grasp the green cube. Without replanning.

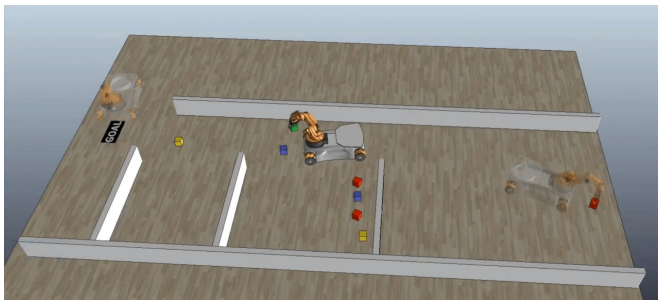


(e) The robot places the object onto the desired location.

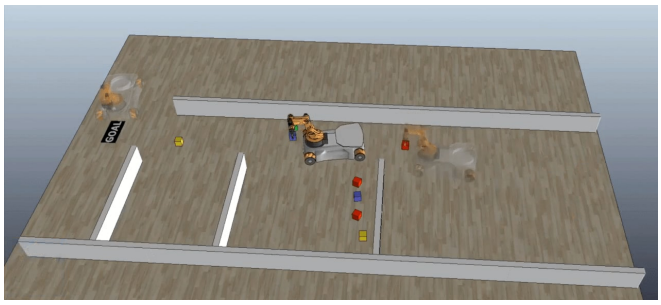
Fig. 11. Execution of a Simple KUKA Youbot experiment.



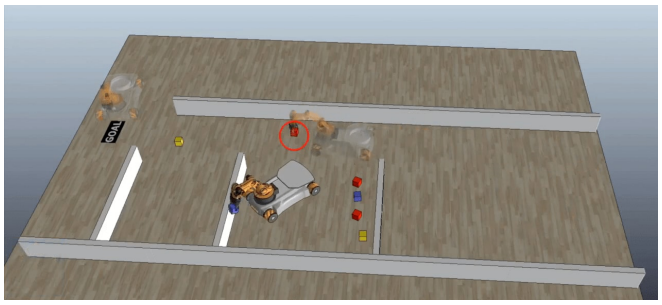
(a) The robot picks the desired object, a green cube.



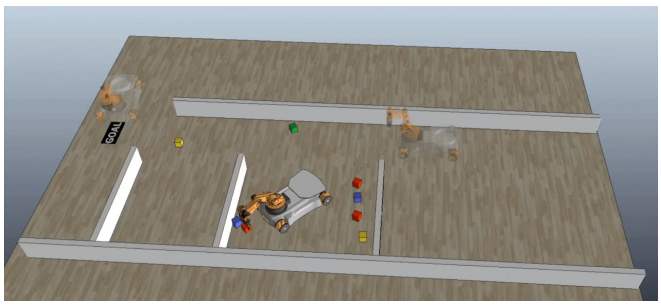
(b) The blue cube obstructs the path to the goal region. The robot drops the green cube in order to pick the blue cube.



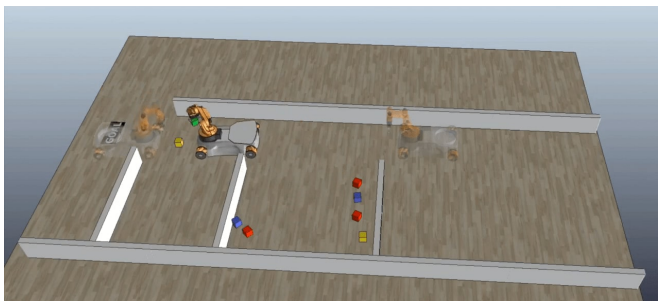
(c) The robot picks the blue cube.



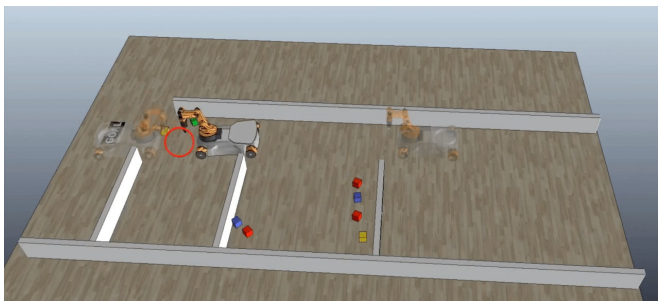
(d) While the robot places the blue cube away from the path to the goal, an external agent places a red cube between the robot and the green cube.



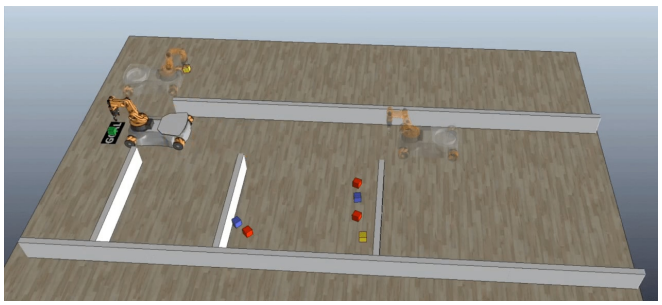
(e) The robot places the red cube away from the path to the goal.



(f) The yellow cube obstructs the path to the goal region. The robot drops the green cube in order to pick the yellow cube.

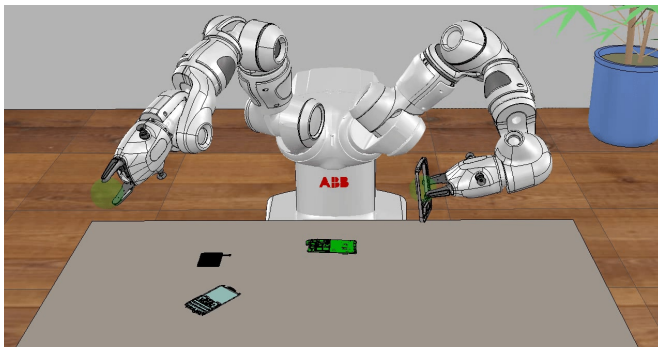


(g) While the robot approaches the yellow cube, an external agent moves the yellow cube away.

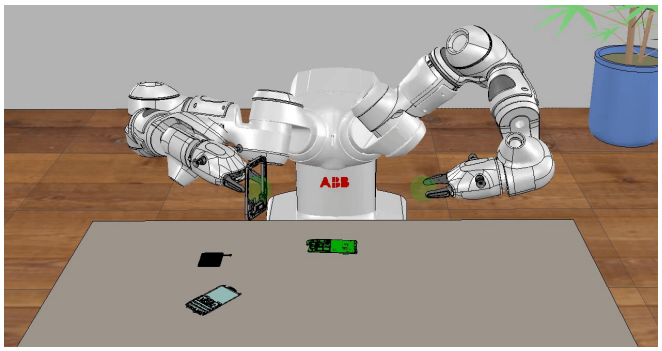


(h) The robot picks the green cube and places it onto the goal region.

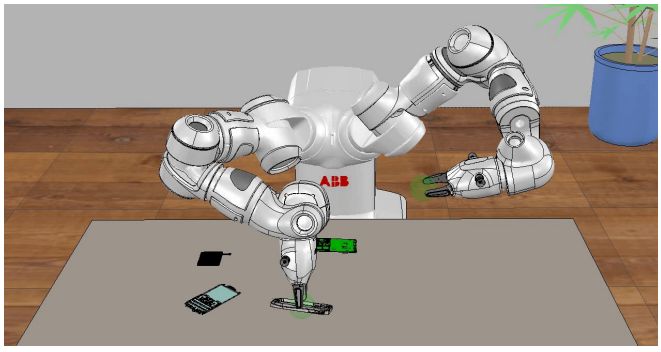
Fig. 12. Execution of a complex KUKA Youbot experiment.



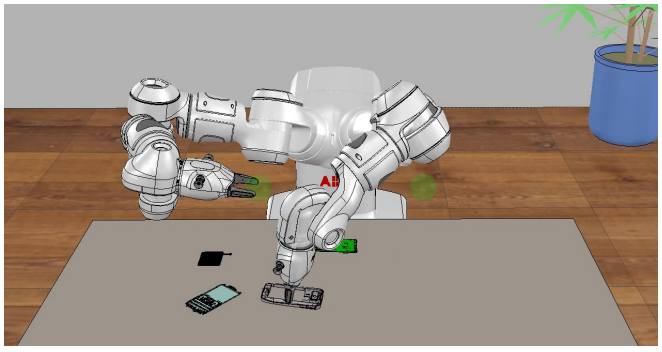
(a) The robot picks the cellphone's chassis. The chassis cannot be assembled with this orientation.



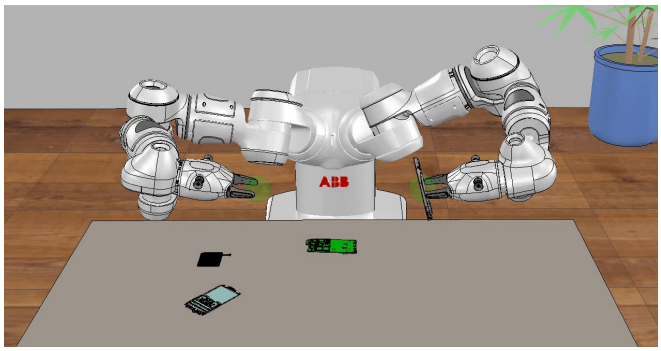
(b) The chassis is handed over the other gripper.



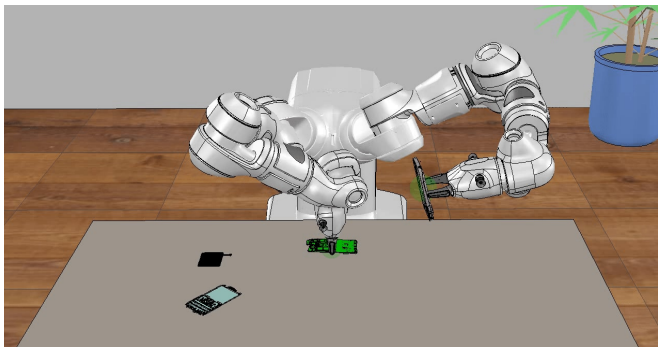
(c) The chassis is placed onto the table with a different orientation than before (the opening part is facing down now).



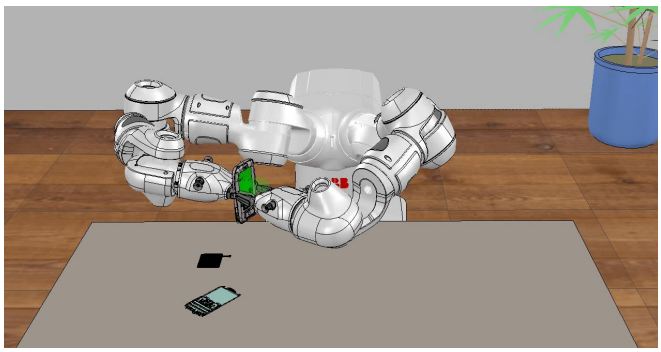
(d) The robot picks the chassis with the new orientation.



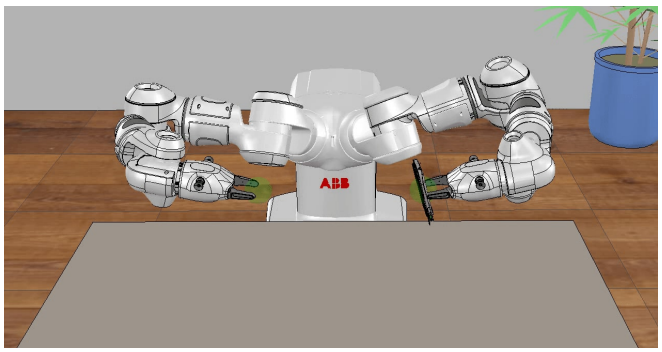
(e) The chassis can be assembled with this orientation.



(f) The robot pick the next cellphone's part to be assembled (the motherboard).



(g) The motherboard and the chassis are assembled.



(h) The robot assembles the cellphone correctly.

Fig. 13. Execution of a ABB Yumi experiment.

REFERENCES

[1] M. Ghallab, D. Nau, and P. Traverso, "The actor's view of automated planning and acting: A position paper," *Artif. Intell.*, vol. 208, pp. 1–17, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2013.11.002

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

[3] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1470–1477.

[4] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, 2005.

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

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

[7] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture and Decision Trees," *to appear in IEEE Transactions on Robotics*, 2016. [Online]. Available: http://michelecolledanchise.com/tro16colledanchise.pdf

[8] M. Colledanchise and P. Ögren, "How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[9] P. Ögren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

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

[11] 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)*, June 2014.

[12] D. Hu, Y. Gong, B. Hannaford, and E. J. Seibel, "Semi-autonomous simulated brain tumor ablation with raven ii surgical robot using behavior tree," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[13] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, "A framework for end-user instruction of a robot assistant for manufacturing," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

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

[15] L. Pena, S. Ossowski, J. M. Pena, and S. M. Lucas, "Learning and Evolving Combat Game Controllers," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 195–202.

[16] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution," *Applications of Evolutionary Computation*, 2011.

[17] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *arXiv preprint arXiv:1504.05811*, 2015.

[18] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Backward-forward search for manipulation planning."

[19] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, and D. Borrajo, "A review of machine learning for automated planning," *The Knowledge Engineering Review*, vol. 27, no. 04, pp. 433–467, 2012.

[20] D. S. Nau, M. Ghallab, and P. Traverso, "Blended planning and acting: Preliminary approach, research challenges," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, pp. 4047–4051. [Online]. Available: http://dl.acm.org/citation.cfm?id=2888116.2888281

[21] J.-C. Latombe, *Robot motion planning*. Springer Science & Business Media, 2012, vol. 124.

[22] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *Int. J. Rob. Res.*, vol. 32, no. 9-10, pp. 1194–1227, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1177/0278364913484072

[23] T. Lozano-Perez, "Automatic planning of manipulator transfer movements," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 10, pp. 681–698, 1981.

[24] T. Lozano-Perez, J. Jones, E. Mazer, P. O'Donnell, W. Grimson, P. Tournassoud, and A. Lanusse, "Handey: A robot system that recognizes, plans, and manipulates," in *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, vol. 4. IEEE, 1987, pp. 843–849.

[25] K. Hauser and V. Ng-Thow-Hing, "Randomized multi-modal motion planning for a humanoid robot manipulation task," *The International Journal of Robotics Research*, vol. 30, no. 6, pp. 678–698, 2011.

[26] J. Barry, L. P. Kaelbling, and T. Lozano-Pérez, "A hierarchical approach to manipulation with diverse actions," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1799–1806.

[27] F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson, "Constraint propagation on interval bounds for dealing with geometric backtracking," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 957–964.

[28] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, "Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 4575–4581.

[29] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 639–646.

[30] M. Levihn, L. P. Kaelbling, T. Lozano-Perez, and M. Stilman, "Foresight and reconsideration in hierarchical planning and execution," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 2013, pp. 224–231.

[31] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1, pp. 5–33, 2001.

[32] J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[33] O. Ilghami, D. S. Nau, H. Muñoz-Avila, and D. W. Aha, "Learning preconditions for planning from plan traces and htn structure," *Computational Intelligence*, vol. 21, no. 4, pp. 388–413, 2005.

[34] N. Abdo, H. Kretzschmar, L. Spinello, and C. Stachniss, "Learning manipulation actions from a few demonstrations," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1268–1275.

[35] T. J. Walsh and M. L. Littman, "Efficient learning of action schemas and web-service descriptions." in *AAAI*, vol. 8, 2008, pp. 714–719.