Reconfigurable Behavior Trees: Towards an Executive Framework Meeting High-level Decision Making and Control Layer Features

Pilar de la Cruz
Dept. of Computer Science
University of Innsbruck
Innsbruck, Austria
pilar.de-la-cruz@uibk.ac.at

Justus Piater
Dept. of Computer Science
and Digital Science Center
University of Innsbruck
Innsbruck, Austria
justus.piater@uibk.ac.at

Matteo Saveriano
Dept. of Computer Science
and Digital Science Center
University of Innsbruck
Innsbruck, Austria
matteo.saveriano@uibk.ac.at

Abstract—Behavior Trees (BTs) constitute a widespread artificial intelligence tool that has been successfully adopted in robotics. Their advantages include simplicity, modularity, and reusability of code. However, Behavior Trees remain a high-level decision making engine; control features cannot easily be integrated. This paper proposes Reconfigurable Behavior Trees (RBTs), an extension of the traditional BTs that incorporates sensed information coming from the robotic environment in the decision making process. We endow RBTs with continuous sensory data that permits the online monitoring of the task execution. The resulting stimulus-driven architecture is capable of dynamically handling changes in the executive context while keeping the execution time low. The proposed framework is evaluated on a set of robotic experiments. The results show that RBTs are a promising approach for robotic task representation, monitoring, and execution.

Index Terms—Decision making, Task representation, Continuous execution monitoring

I. INTRODUCTION

Modern robotic agents demand increased flexibility in the way the assigned task is represented and executed. Indeed, autonomous robotic systems need to be capable of dealing with sensing and planning operations at low cost, as well as monitoring actions in a goal-oriented fashion [1]. Behavior Trees (BTs) constitute a powerful tool for task switching and decision making, and are receiving increasing attention in the robotics community [2]–[4]. The reason for this growing attention mostly lie in the fact that BTs are self-explanatory, modular, facilitate code reuse, and are simple to design. A BT is built combining a limited number (six) of node types. This greatly simplifies the design of new BTs, makes them *human-readable*, and eases the formal verification of the generated task plan without curbing their expressive power [3]. Modularity has been exploited in robotics in both hardware [5], [6] and software [7]. BTs are modular in the sense that each subtree may be seen as a subblock that may be added or

replaced by any other subblock. This makes the code reusable for different applications and further simplifies the design of new BTs.

However, BT engines are not designed to operate within the sense-plan-act paradigm, nor do they provide an optimized trade-off between reactiveness and execution cost for low-level control. Also, BTs may easily grow when the number of actions and conditions needed for closing the execution loop increases. Moreover, continuous monitoring of the task execution as well as online resolution of possible ambiguities in the task plan are typically not supported by BTs. This limits the applicability of BTs in dynamic or uncertain environments. In order to overcome these limitations, a robotic executive framework has to: *i)* ensure low complexity in terms of cost and implementation when dealing with task executions, *ii)* make a connection between low-level stimuli and high-level decision making, and *iii)* perform a continuous and goal-oriented planning.

This paper proposes an executive framework that meets the robotic task requirements by combining the planning capabilitites of BTs with attentional mechanisms for control features [8], [9]. The proposed *Reconfigurable Behavior Trees (RBTs)* exploit the high modularity of traditional BTs to define a tree structure that can be reconfigured at runtime, i.e. dynamically during the task execution, by adding and/or removing parts of the tree. The reconfiguration is ruled by environmental stimuli corresponding to changes in the sensed information and by the successful execution of goal-directed actions. This paper presents a formal definition of the RBT framework and evaluates its performance in scheduling robotic tasks. To summarize, our contribution is two-fold:

- We extend the BT formalism to incorporate continuous information in the decision process. This makes it possible to monitor the execution and react to unexpected changes in the environment.
- We propose an algorithm to dynamically allocate and deallocate tree branches. Subtrees are stored as JSON (JavaScript Object Notation) schemas, a format that is

supported by most existing databases.

Section II presents related work. In Sec. III, we describe the proposed approach. Section IV presents simulation experiments and evaluates the results. Section V states the conclusions and proposes further extensions.

II. RELATED WORK

Finite state machines have been widely used in different areas of computer science and engineering. A finite state machine provides a basic mathematical computational model consisting of a definite set of states, transitions, and events. Finite state machines are flexible and easy to design, but as soon as the system grows in complexity, a reactivity/modularity trade-off problem arises and the approach becomes impractical.

Decision trees [10], the subsumption architecture [11], and sequential behavior composition [12] are widely-used approaches to decision making and task execution in robotics. A decision tree is an analytical decision support tool consisting of control structures and predicates located in the leaves and internal nodes, respectively, which map the possible outcomes of a series of choices. One motivation for their use is their "white-box" nature, i.e. decisions can be intuitively explained, they are simple to visualize, and may be easily implemented. However, decision trees lack robustness to noisy perceptual data, and their size rapidly increases in complex scenarios. The subsumption architecture relies on having a number of controllers in parallel which are ordered with different priorities so that each one is allowed to output both actuator commands and a binary value, signaling if the control of the robot is active or inactive. In sequential behavior composition, the behavior is driven by local controllers. The state space is split into cells, corresponding to the basin of attraction of each controller. The task of each controller is to drive the system into the basin of attraction of another controller that is closer to the overall goal state.

Colledanchise and Ögren [3] have shown that BTs represent an elegant generalization of finite state machines, decision trees, the subsumption architecture, and sequential behavior composition. Several fixed-logic control models have emerged that extend the functionalities of BTs and attempt to overcome their limitations in highly dynamic environments. For instance, Conditional BTs [13] extend traditional BTs to monitor the execution of an action considering logic pre- and postconditions. The work in [14] proposes to cast an automated plan created by a hierarchical task network planner [15], [16] into executable BTs. In this way, BTs provide reactiveness and modularity, whereas the planner is responsible for the deliberative behavior of the robot. Along the same lines, other work [4], [17], [18] explores different ways of synthesizing a BT using a planner. Automatic synthesis is employed to produce BTs with guaranteed performance [19] or safety [20]. Other approaches extend BTs by integrating models where domain knowledge can be learned automatically, for instance using reinforcement learning [21], [22], genetic programming [23], or imitation

TABLE I
TYPES OF BT NODES AND THEIR RETURN STATUS

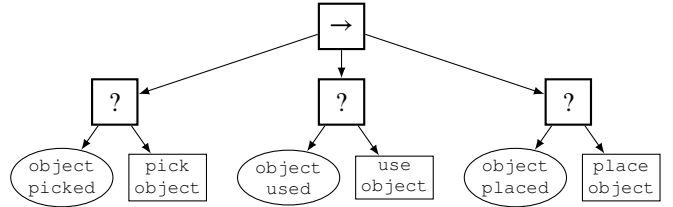| Type	Symbol	Success	Failure
Fallback/Selector	?	One child succeeds	All children fail
Sequence	→	All children succeed	One child fails
Parallel	⇉	$>M$ children succeed	$>N-M$ children fail
Decorator	◇	Custom	Custom
Action	▭	Upon completion	Impossible to complete
Condition	◯	True	False

Fig. 1. A BT representing a pick-use-place task. → is a Sequence node; ? are Fallback nodes. Ellipses define condition nodes and rectangles define action nodes.

learning [24]. Learning techniques can overcome the limitations of classical planners that require significant engineering effort [25]. Finally, the work in [26] attempts to map the practical solutions developed for action sequencing in real-time strategy games to robotic applications.

Characteristics like human-readability, expressivity, modularity, and reusability make BT-based techniques popular and unquestionably attractive. However, critical aspects that these approaches do not cover are the connection to the physical executive state and the possibility of ambiguities in the decision making. Their rigidity is not desirable for systems with strong perceptual constraints, like robots intended to interact with the environment and able to flexibly behave and react to unexpected events. To tackle this issue, some work exploits attentional mechanisms for visual task learning [27], for cognitive control of humanoid robots [8], and for flexible orchestration of cooperative tasks [9]. Inspired by the way humans monitor and orchestrate task execution [28], [29], the attentional framework in [9] loads tasks from a long-term memory and instantiates them in a working memory using a mechanism analogous to that used in Hierarchical Task Network planning [15]. Additionally, continuous sensory data are exploited to solve any ambiguities in the task plan and to quickly react to environmental changes. This attentional system has been effectively integrated in an imitation learning framework to learn, plan, and monitor complex robotic tasks [30]–[33].

In this work, we take the best of these two worlds and propose RBTs, an executive framework that combines the human-readability, modularity, and reusability of BTs with the additional flexibility offered by attention-based cognitive architectures.
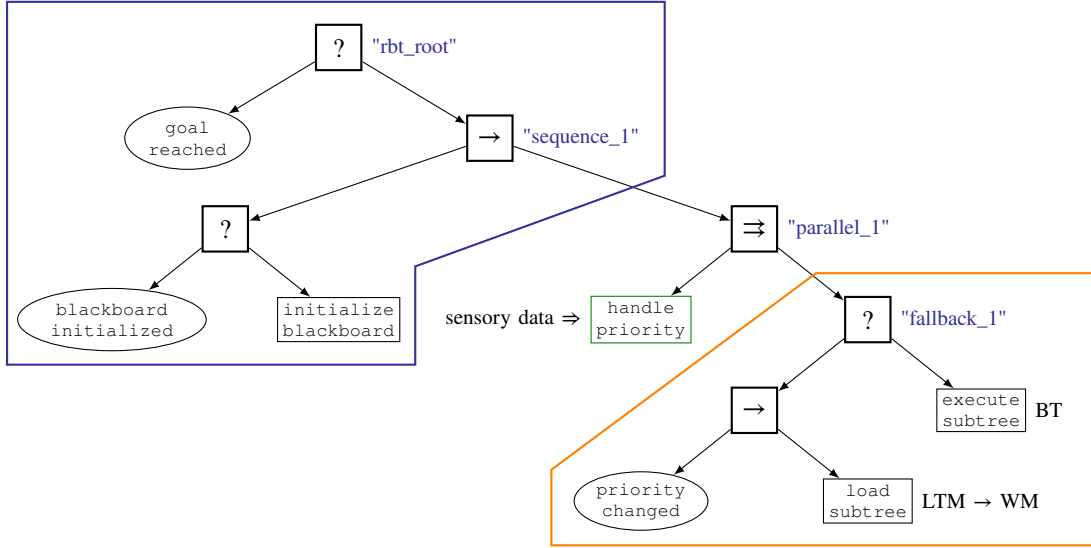
Fig. 2. The generic RBT. The branch of the tree surrounded by the blue polygon allows execution to be terminated after a global goal is reached and is the same for RBT and BT. The green action node is the *Emphasizer* that modifies the priority of each subtree considering the environmental stimuli. The branch of the tree surrounded by the orange polygon is dynamically allocated/deallocated by the *Instantiator* every time the subtree priority order changes. The action node `execute subtree` contains small BTs like the one in Fig. 1. The blue labels are the node names used in Listing 1.

III. RECONFIGURABLE BEHAVIOR TREES

A. Behavior Trees

A BT is a graphical model language to control the actions (or behaviors) of an autonomous agent and execute a task. A BT consists of a tree structure containing a combination of the six types of nodes shown in Table I. These types are divided into two categories: *control flow* and *execution* nodes. The four types of control flow nodes are Sequence, Fallback/Selector, Decorators, and Parallel nodes. The two types of execution nodes are Condition and Action nodes. Each type of node returns a *running* state during the execution and *success* or *failure* after the execution. The execution of a BT is possible by periodically traversing the tree from the root node to all child nodes from left to right. The traversing mechanism is periodically activated by sending a signal called "tick". Each child node responds to this signal according to its own type and to the return state of the other nodes. Table I describes the behavior of each type of node for the Success and Failure cases. The running state will behave in a similar fashion.

A minimal example of a BT applied to a pick-use-place object task is shown in Fig. 1. The root being a Sequence node, the BT is executed sequentially from left to right. If the condition `object picked` has not been fulfilled, i.e. the node returns `False`, the action node `pick object` enters the *Running* state. The action node returns success upon successful completion of the pick action. As a consequence, the first (far left) Fallback node also returns success and the traversal mechanism enters the second (middle) Fallback node. This procedure is repeated until all the Fallback nodes return success, indicating that the task has been successfully completed.

B. RBT components

The generic RBT, depicted in Fig. 2, is a BT enriched with extra functionalities that permits the continuous monitoring of environmental stimuli and the dynamic reconfiguration of the tree to execute. Interestingly, those functionalities are implemented using the same six types of nodes considered in the standard BTs (see Table I), leaving unaffected the design simplicity typical of BTs. Traversing the RBT from the root (Fallback) node, we first check if the end goal of the task is fulfilled. If not, we check if the blackboard is initialized and eventually run `initialize blackboard`. The blackboard is a mechanism used in BTs to store and update runtime variables and to make them accessible by each node in the tree. In the RBT framework, the blackboard contains the logical pre- and postconditions used to regulate the task execution and to determine when the task goal is fulfilled, the sensed information used to set the priorities of each subtree, and the current value of the subtrees priorities. The blackboard is dynamically updated and greatly simplifies the communication between nodes by handling the concurrent access in a transparent and thread safe manner. We would like to point out that this part of the RBT, surrounded by a blue polygon in Fig. 2, allows execution to be terminated after a global goal is reached. A similar branch has to be introduced in the standard BT to successfully terminate the task and therefore is does not introduce extra nodes in the RBT. Once the blackboard is initialized the right branch of the RBT is traversed and two parallel processes start. On one side, sensory data are processed to determine the priority of the S available subtrees. On the other side, the most emphasized subtree is instantiated and executed asynchronously with respect to the perceptual input. The instantiation mechanism is dynamic and

allows for flexible task orchestration.

Compared to a BT, the RBT has the following extra components:

1) A *Long-Term Memory (LTM)* and a *Short-Term* or *Working Memory (WM)* that are typical of attention-based control frameworks [9].
2) A priority handler, namely the *Emphasizer*, that computes the highest-priority task considering the sensed information and logical pre- and postconditions.
3) An *Instantiator* that accesses the LTM and casts the subtask into the corresponding subtree loaded in the WM. The Instantiator enables the RBT reconfiguration capabilities by dynamically loading and instantiating the subtree with higher priority.

The distinctive components of RBTs are detailed in the rest of this section.

C. LTM and WM

The LTM can be considered a database that contains all subtasks that the robot is able to execute. A typical subtask is the pick-use-place object task described in Sec. III-A. In order to store and retrieve subtasks from the LTM, we propose a common representation of the 4 control flow nodes in Table I. The generic control flow node $\mathcal{B}$ is represented as a quadruple

$$\mathcal{B} = (l, t, c, p) \tag{1}$$

where l is the unique name (label) of the node, t is one of the types in Table I, c is a list of children, and p is a list of *parameters* like the priority value or pre- and postconditions.

In principle, it is possible to represent also the 2 execution nodes in Table I as the quadruple defined in (1). However, we found a more convenient way of exploiting the fact that execution nodes correspond to leaves in the BT. In more detail, action nodes are specified only in the children list of the parent node, while the condition nodes are used to represent the pre- and postconditions that are listed in the parameter list p. The successful execution of an action also changes the state of the relative postcondition, while the preconditions of an action are changed by other nodes in the tree.

Following this representation, the LTM can be conveniently organized in JSON schemas. As shown in Listing 1, the root of a tree is identified by the key word `root` in its name (`rbt_root`). Actions are simply listed as children of a node and are identified by the string `A(action_name)`. For preconditions, we use the notation `C_ij` where `i` is the child number and `j` indicates the *j*-th condition. Postconditions (or goals) are identified by `G_ij` where `i` is the child number and `j` indicates the *j*-th condition. It is worth noticing that the described representation contains all the information needed to instantiate an executable BT and that no further JSON schemas are needed to describe the leaves of the BT.

The *Instantiator* is responsible for loading the task from the LTM and creating an instance of the BT to execute in the WM. This procedure is summarized in Algorithm 1. Given the task name (root of the BT), the Instantiator first loads the JSON schemas describing the task (line 2 in Algorithm 1).

Listing 1. JSON schemas representing the generic RBT in Fig. 2.

```
{
  "name":"rbt_root",
  "type":"fallback",
  "children": ["sequence_1"],
  "params": ["G_11", "goal reached"]
},
{
  "name":"sequence_1",
  "type":"sequence",
  "children": ["A(initialize blackboard)",
              "parallel_1"],
  "params": ["G_11",
            "blackboard initialized"]
},
{
  "name":"parallel_1",
  "type":"parallel",
  "children": ["A(handle priority)",
              "fallback_1"],
  "params": [""]
}
{
  "name":"fallback_1",
  "type":"fallback",
  "children": ["A(load subtree)",
              "A(execute subtree)"],
  "params": ["C_11", "priority changed"]
}
```

Starting from the root, the BT is built by iteratively expanding the nodes until the leaves are reached (lines 4 to 19). The current JSON schema is converted into the BT node specified in the `type` field and attached to the current tree (line 5). Pre- and postcondtions are handled using a modified version of the Planning and Acting PA-BT approach in [2] that allows multiple postconditions. In this approach, a postcondition is transformed into a Condition node (line 7) that is connected to the rest of the tree via a Fallback node ($\mathcal{T}_{fal}$). In this way, execution ends when the postcondition becomes `True`. The case of a single postcondition is handled in lines 8–9. In case of multiple postconditions, a Sequence node is created with all the postconditions attached (line 11). In this way, the postconditions are sequentially verified. The Sequence node containing all the postconditions is then attached to $\mathcal{T}_{fal}$ (line 12). In both cases the generated Fallback node is attached to the current tree (line 18). Lines 14–17 handle Action nodes with associated preconditions. As for the postconditions, the preconditions are considered as Condition nodes (line 16). Action and Condition nodes are then connected to a Sequence node that is attached to the current tree (lines 17–18). In this way, an Action is executed only if all the preconditions are `True`. As a final note, the functions SEQUENCENODE and FALLBACKNODE in Algorithm 1 return an empty subtree if the input is empty, while SEQUENCENODE({}, $\mathcal{A}$) returns the Action nodes $\mathcal{A}$.

Algorithm 1 Load and instantiate a BT

1: **function** INSTANTIATESUBTREE(*l*)   ▷ *l*: subtask name
2:     schemaList ← GETTASKFROMLTM(*l*)
3:     $\mathcal{T} \leftarrow \{\}$ ▷ empty BT
4:     **for** schema in schemaList **do**
5:         $\mathcal{T} \leftarrow$ SCHEMATONODE($\mathcal{T}$, schema)
6:         postC ← GETPOSTCONDITIONS(schema)
7:         $\mathcal{C} \leftarrow$ CONDITIONNODES(postC)
8:         **if** $\mathcal{C}$.length == 1 **then**
9:             $\mathcal{T}_{fal} \leftarrow$ FALLBACKNODE($\mathcal{C}$)
10:        **else**
11:            $\mathcal{T}_{seq} \leftarrow$ SEQUENCENODE($\mathcal{C}$)
12:            $\mathcal{T}_{fal} \leftarrow$ ATTACHSUBTREE($\mathcal{T}_{seq}$)
13:        **end if**
14:        a, preC ← GETACTIONS(schema)
15:        $\mathcal{A} \leftarrow$ ACTIONNODES(a)
16:        $\mathcal{C} \leftarrow$ CONDITIONNODES(preC)
17:        $\mathcal{T}_{seq} \leftarrow$ SEQUENCENODE($\mathcal{C}$, $\mathcal{A}$)
18:        $\mathcal{T} \leftarrow$ ATTACHSUBTREE($\mathcal{T}_{fal}$, $\mathcal{T}_{seq}$)
19:    **end for**
20:    **return** $\mathcal{T}$
21: **end function**

D. Subtree priority

The modularity of standard BT allows a complex task (tree) to be decomposed into subtasks (subtrees). For instance, a stacking task can be decomposed as a combination of pick and place subtasks. However, in standard BT, the execution order of each subtask is predefined. Changing the execution order depending on discrete values is possible, but requires extra branches in the BT. Changing the execution order by monitoring continuous values like sensory data is typically not supported.

In contrast, RBTs exploit logical pre- and postconditions, as well as continuous sensory data, to monitor the task execution. As already mentioned, a complex task is divided into subtrees. We assign pre- and postconditions to the root of each subtree. Therefore, a specific subtree is *active* if all the preconditions are satisfied while the postconditions are not. A subtree correctly terminates by setting the postconditions. At each tick, the Emphasizer accesses the blackboard and looks for active subtrees. An execution conflict occurs every time more than one active subtree exists. In this case, we exploit a priority-based mechanism to dynamically decide the subtask to execute. We define the priority of a subtree as the runtime prominence for the execution of a specific subtask. The priority is a real value, normalized between 0 and 1, which tells the Instantiator which subtask to load and transform into an executable BT. In this work, the priority ϵ is defined as

$$\epsilon(\theta) = \begin{cases} 1 & \text{if } \theta \leq \theta_{\min} \\ \dfrac{\theta - \theta_{\max}}{\theta_{\min} - \theta_{\max}} & \text{if } \theta_{\min} < \theta < \theta_{\max} \\ 0 & \text{if } \theta \geq \theta_{\max} \end{cases} \quad (2)$$

where θ is a continuous value coming from sensory data, and the thresholds $\theta_{\min}$ and $\theta_{\max}$ are tunable parameters. In this work, θ represents the inverse of the distance from objects to manipulate, but other choices are possible.

E. Task execution and monitoring

The generic RBT, as the one depicted in Fig. 2, is a goal-oriented tree that successfully terminates only if a certain goal is achieved (the `goal reached` condition becomes `True`). In our implementation, the goal of the RBT is achieved if the postconditions of all the subtrees are satisfied. As already mentioned, the execution of the RBT is achieved by periodically traversing the tree top to bottom and left to right (tick function). At each tick, the `goal reached` condition is tested. If `goal reached` is `False`, we check that the blackboard is initialized and then enter a Parallel node. Executing the Emphasizer and the subtree in parallel is convenient because sensory data and task execution are, in general, asynchronous processes. The Parallel node is designed to successfully terminate if both children are successfully executed. Since the Action node `handle priority` is always in the running state, the Parallel node cannot terminate. This implies that the RBT successfully terminates if and only if the goal is reached.

If new sensory data arrive or a subtree postcondition changes, the Emphasizer recomputes the priority of the subtrees and sets the `priority changed` flag. This triggers the Instantiator that preempts the current execution, deallocates the current subtree, and instantiates the subtree with highest priority. As already mentioned, this branch of the tree—the branch inside the orange polygon in Fig. 2—is dynamically allocated at each tick. The dynamic allocation is needed for the correct execution of the task. To better understand this point, consider what happens if `execute subtree` returns `True`. In this case, the active subtask has been successfully executed and the Fallback node (`fallback_1`) also returns `True` (see Table I). With a static branch, the tick would not enter `fallback_1` anymore, letting the remaining active subtasks unexecuted. With a dynamic branch, instead, the return state of `fallback_1` is reset and the active subtask with highest priority is correctly instantiated and executed.

IV. EVALUATION

We evaluate RBTs in the sorting task shown in Fig. 3 where the robot has to pick 3 colored boxes (red, blue, and green) from a table (Fig. 3(a)) and place them at specific locations in a storage area (Fig. 3(b)). The scenario is simulated in CoppeliaSim [34] using the Panda robot model provided by Gaz et al. [35]. We consider two different case studies and compare the performance of RBTs and BTs in terms of execution time and tree complexity—the number of nodes in the tree. Note that the execution time is computed by summing up the tick times until the goal is reached and without considering the time spent by the robot to perform the commanded actions. BTs are implemented using the open-source Python
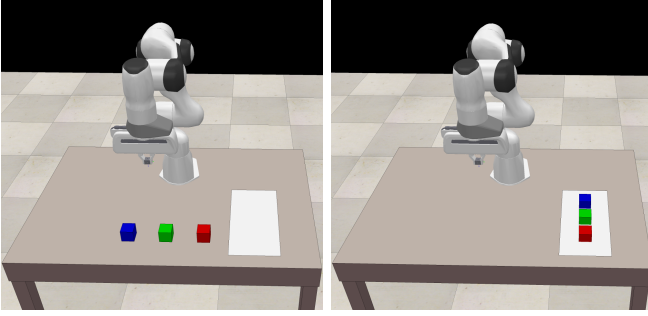
(a) Possible initial configuration.   (b) Desired final configuration.

Fig. 3. The sorting task where the robot has to pick 3 boxes from the table and place them at a specific location in the white storage area.
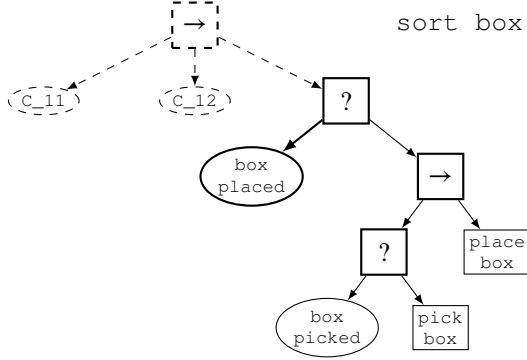

Fig. 4. The BT used to pick a box from the table and place it in the storage area. This BT is compactly represented by the `execute subtree` node in the RBT of Fig. 2. The RBT uses the dashed nodes to impose the preconditions in case study 1, while in case study 2 they are omitted.
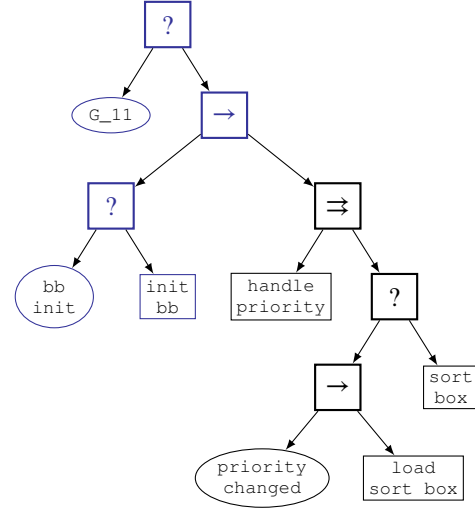

Fig. 5. The RBT used to plan the sorting task. In case study 1, `sort box` is subject to preconditions to constrain the execution. The blue nodes are common to BTs and RBTs.

TABLE II
COMPARISON BETWEEN RBT AND STANDARD BT.

Method	Case	# Nodes	Execution Time (ms)
RBT	1	19 - 22	507.04
BT	1	27	503.24
RBT	2	19	505.17
BT	2	151	819.78

library `py_tree` [36]. RBTs are also implemented in Python exploiting the basic BT nodes provided by `py_tree`.

Figure 4 shows the BT used to plan box sorting subtasks. The solid nodes are common between BTs and RBTs, while the dashed nodes are exploited by RBTs to enforce a certain execution order specified by a set of preconditions {`C_11`, `C_12`}. As discussed in Sec. III, RBTs dynamically attach the subtree in Fig. 4 to the *static* tree in Fig. 2. Moreover, the standard BT is endowed with 6 extra nodes (contained in the blue polygon in Fig. 2) to monitor the execution of the task and successfully terminate after a goal is reached. The task is fulfilled when all the boxes are sorted in the storage area, as indicated by the task goal `G_11` = b_box placed ∧ g_box placed ∧ r_box placed. The thresholds used to compute the priority in (2) are empirically set to $\theta_{min} = 0.05\,\text{m}$ (the length of the box side) and to $\theta_{max} = 1\,\text{m}$ (the maximum distance that still allows grasping a box). We compare RBT and BT in terms of number of nodes and execution time measured assuming that action nodes directly return `True` without entering the running state.

1) Case study 1: The goal of this experiment is to compare RBTs and BTs in a situation that favours the BT, i.e. when the task plan is rigid, no ambiguities are possible, and no external disturbances occur. In this case, we assume that the

boxes need to be sorted in a specific order: First we pick and place the blue box b_box, then the green box g_box, and finally the red box r_box. This sorting order has been arbitrarily decided and does not affect the obtained results. The sorting task successfully terminates when the task goal `G_11` is reached. The BT used to perform the sorting task is shown in Fig. 6(a), where the 3 Action nodes sort *box_name* compactly represent the BT in Fig. 4 (only the solid nodes are considered). As listed in Table II, the BT of Fig. 6(a) has 27 nodes. The RBT used to plan this task is shown in Fig. 5 and is almost identical in the two case studies. The only difference is in the `sort box` Action node. In case study 1, we exploit preconditions (dashed nodes in Fig. 4) to impose a certain execution order. More specifically, the Action node `sort box` is allocated by the *Instantiator* that, at runtime, dynamically instantiates a specialized version of the sorting task where the generic box is replaced by the one with highest priority. As described in Sec. III-D, the priority of a subtask depends on logical precondition and continuous stimuli. In this case, external stimuli play no role and the execution order is fully determined by the preconditions. In particular, sort b_box has no preconditions and is the first to be executed. sort g_box has `C_11` = b_box placed as precondition, while sort r_box has `C_11` = b_box
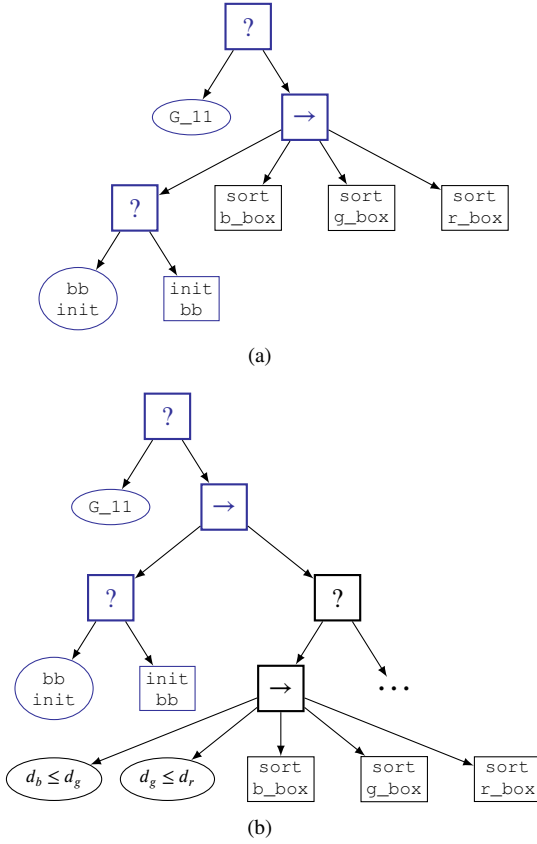
(a)

(b)

Fig. 6. The BT used to plan the sorting task of case study 1 (a) and case study 2 (b). The blue nodes are common to BTs and RBTs. The variable d_i, $i = b, r, g$, indicates the distance between the i_box and the robot. Due to the limited space, we only show a partial BT (b). In particular, the stump containing the black sequence node and its children (24 nodes in total) handles the cases $d_b \leq d_g$ and $d_g \leq d_r$. Five similar stumps are needed to handle all possible combinations of box distances and are compactly indicated here by the symbol $\cdots$.

placed and C_12 = g_box placed as preconditions. This guarantees that the sorting task is executed in the desired sequential order. As listed in Table II, the RBT has a variable number of nodes (19 to 22). This is because the sort node has a variable number of preconditions for each box. Even if in case study 1 the RBT does not show its full potential, we still have a reduction of the number of nodes in the tree (22 nodes in RBT in the worst case, 27 in the BT). The time to execute the BT is slightly smaller in this case, but the difference is negligible (less than 4 ms).

2) Case study 2: This experiment is designed to show the benefits of the priority-based task execution introduced by RBTs. The scenario is the same as in case study 1 but here the boxes are sorted without a predefined order. In principle, there are 6 possibilities (plans) to pick the 3 boxes in Fig. 3(a) and place them in the goal configuration shown in Fig. 3(b). Sort b_box, r_box, g_box and sort b_box, g_box, r_box are 2 of the 6 of feasible plans. We use the robot-box distances to determine which plan the robot executes. In particular, we always sort first the box closest to the gripper.. The RBT used to plan this task is the

same as in case study 1, except for the sort box Action node that has no general preconditions. Therefore, the RBT always has 19 nodes (see Table II). At each time, all the boxes on the table are eligible to be sorted. These execution conflicts are managed in RBTs using the task priorities computed with (2). Hence, the *Instantiator* is free to allocate the subtree with highest priority, i.e. to start sorting the closest box. Once a box, for instance the blue one, is placed in the storage area, the postcondition b_box placed becomes True and the *Emphasizer* removes sort b_box from the list of active nodes (see Sec. III-D). This implies that sort b_box is not instantiated in future ticks, letting the robot sort the other boxes and successfully complete the task. In order to reach a similar level of flexibility with standard BTs, we need many more nodes to represent the 6 feasible plans in one tree and a more complicated control flow logic to determine the plan to execute based on robot-box distances. A possible solution that requires 151 nodes is sketched in Fig. 4(b). In this case, RBTs require $\approx 85\%$ less nodes, which clearly makes the RBT easier to visualize, and a reduction of the execution time of $\approx 38\%$.

V. CONCLUSION AND FUTURE WORK

We introduced Reconfigurable Behavior Trees, a novel framework that combines high-level decision making and continuous monitoring and control features. By combining the expressivity and modularity of standard behavior trees with the flexibility of attentional execution monitoring, RBTs allow the AI agent to perform actions in a robust and versatile way, while being capable of adjusting its behavior based on continuous input from the environment. The proposed framework has been tested in a sorting task and compared with standard BTs in terms of tree complexity and computation time. The evaluation shows that RBTs outperform standard BTs, especially when the task plan is not rigidly defined and ambiguities in the execution need to be solved.

This paper focused on theoretical description of RBTs and comparisons with traditional BTs. In [37], we combined RBTs with a reactive motion generation approach to correct the robot trajectory on-the-fly without a computationally expensive motion re-planning step. The preliminary results shown in this paper and in [37] are promising and suggest that RBTs are better suited to dynamic and uncertain environments than BTs. Therefore, we expect that RBTs can significantly contribute to the deployment of social robotics applications. This clearly requires further development of the proposed framework. Among others, the RBT framework has to be tested in human-robot interaction scenarios. Uncertainty introduced by the human in the loop can be estimated using, for instance, neural networks [38] and can explicitly be considered in the decision process.

REFERENCES

[1] J. López, A. Santana-Alonso, and M. Díaz-Cacho Medina, "Formal verification for task description languages. a petri net approach," *Sensors*, vol. 19, no. 22, p. 4965, 2019.

[2] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.

[3] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," *Transactions on robotics*, vol. 33, no. 2, pp. 372–389, 2016.

[4] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *International Conference on Intelligent Robots and Systems*, 2017, pp. 6793–6800.

[5] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular self-reconfigurable robot systems [grand challenges of robotics]," *IEEE Robotics & Automation Magazine*, vol. 14, no. 1, pp. 43–52, 2007.

[6] P. Moubarak and P. Ben-Tzvi, "Modular and reconfigurable mobile robotics," *Robotics and autonomous systems*, vol. 60, no. 12, pp. 1648–1663, 2012.

[7] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[8] K. Kawamura, S. M. Gordon, P. Ratanaswasd, E. Erdemir, and J. F. Hall, "Implementation of cognitive control for a humanoid robot," *International Journal of Humanoid Robotics*, vol. 5, no. 4, pp. 547–586, 2008.

[9] R. Caccavale and A. Finzi, "Flexible task execution and attentional regulations in human-robot interaction," *Transactions on Cognitive and Developmental Systems*, vol. 9, no. 1, pp. 68–79, 2016.

[10] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[11] R. Brooks, "A robust layered control system for a mobile robot," *Journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.

[12] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, "Sequential composition of dynamically dexterous robot behaviors," *The International Journal of Robotics Research*, vol. 18, no. 6, pp. 534–555, 1999.

[13] E. Giunchiglia, M. Colledanchise, L. Natale, and A. Tacchella, "Conditional behavior trees: Definition, executability, and applications," in *International Conference on Systems, Man and Cybernetics*, 2019, pp. 1899–1906.

[14] J. A. Segura-Muros and J. Fernández-Olivares, "Integration of an automated hierarchical task planner in ros using behaviour trees," in *International Conference on Space Mission Challenges for Information Technology*, 2017, pp. 20–25.

[15] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "Shop2: An HTN planning system," *Journal of artificial intelligence research*, vol. 20, pp. 379–404, 2003.

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

[17] X. Neufeld, S. Mostaghim, and S. Brand, "A hybrid approach to planning and execution in dynamic environments through hierarchical task networks and behavior trees," in *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[18] M. Lan, S. Lai, T. H. Lee, and B. M. Chen, "Autonomous task planning and acting for micro aerial vehicles," in *2019 IEEE 15th International Conference on Control and Automation (ICCA)*, 2019, pp. 738–745.

[19] C. Paxton, N. Ratliff, C. Eppner, and D. Fox, "Representing robot task plans as robust logical-dynamical systems," in *International Conference on Intelligent Robots and Systems*, 2019, pp. 5588–5595.

[20] T. G. Tadewos, L. Shamgah, and A. Karimoddini, "Automatic safe behaviour tree synthesis for autonomous agents," in *IEEE Conference on Decision and Control*, 2019, pp. 2776–2781.

[21] Q. Zhang, Q. Yin, and Y. Hu, "Modeling cgfs behavior by an extended option based learning behavior trees," in *International Conference on Cybernetics and Intelligent Systems and Conference on Robotics, Automation and Mechatronics*, 2017, pp. 260–265.

[22] X. Zhu, "Behavior tree design of intelligent behavior of non-player character (npc) based on unity3d," *Journal of Intelligent & Fuzzy Systems*, vol. 37, no. 5, pp. 6071–6079, 2019.

[23] Q. Zhang, J. Yao, Q. Yin, and Y. Zha, "Learning behavior trees for autonomous agents with hybrid constraints evolution," *Applied Sciences*, vol. 8, no. 7, p. 1077, 2018.

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

[25] 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.

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

[27] A. Borji, M. N. Ahmadabadi, B. N. Araabi, and M. Hamidi, "Online learning of task-driven object-based visual attention control," *Image and Vision Computing*, vol. 28, no. 7, pp. 1130–1145, 2010.

[28] D. A. Norman and T. Shallice, "Attention to action," in *Consciousness and self-regulation*. Springer, 1986, pp. 1–18.

[29] R. P. Cooper and T. Shallice, "Hierarchical schemas and goals in the control of sequential behavior," *Psychological Review*, vol. 113, no. 4, pp. 887–931, 2006.

[30] R. Caccavale, M. Saveriano, G. A. Fontanelli, F. Ficuciello, D. Lee, and A. Finzi, "Imitation learning and attentional supervision of dual-arm structured tasks," in *Joint International Conference on Development and Learning and Epigenetic Robotics*, 2017, pp. 66–71.

[31] R. Caccavale, M. Saveriano, A. Finzi, and D. Lee, "Kinesthetic teaching and attentional supervision of structured tasks in human–robot interaction," *Autonomous Robots*, vol. 43, no. 6, pp. 1291–1307, 2019.

[32] M. Saveriano, M. Seegerer, R. Caccavale, A. Finzi, and D. Lee, "Symbolic task compression in structured task learning," *International Conference on Robotic Computing*, pp. 171–176, 2019.

[33] A. Agostini, M. Saveriano, D. Lee, and J. Piater, "Manipulation planning using object-centered predicates and hierarchical decomposition of contextual actions," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 5629–5636, 2020.

[34] E. Rohmer, S. P. N. Singh, and M. Freese, "Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework," in *International Conference on Intelligent Robots and Systems*, 2013, pp. 1321–1326.

[35] C. Gaz, M. Cognetti, A. Oliva, P. Robuffo Giordano, and A. De Luca, "Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization," *Robotics and Automation Letters*, vol. 4, no. 4, pp. 4147–4154, 2019.

[36] D. Stonier, "py_trees documentation," https://py-trees.readthedocs.io/_/downloads/en/devel/pdf/, online; accessed on 25 March 2020.

[37] M. Saveriano and J. Piater, "Combining decision making and dynamical systems for monitoring and executing manipulation tasks," *e & i Elektrotechnik und Informationstechnik*, 2020.

[38] H. M. D. Kabir, A. Khosravi, M. A. Hosen, and S. Nahavandi, "Neural network-based uncertainty quantification: A survey of methodologies and applications," *IEEE Access*, vol. 6, pp. 36 218–36 234, 2018.