Building the Foundation of Robot Explanation Generation Using Behavior Trees

ZHAO HAN, DANIEL GIGER, and JORDAN ALLSPAW, University of Massachusetts Lowell, USA
MICHAEL S. LEE and HENNY ADMONI, Carnegie Mellon University, USA
HOLLY A. YANCO, University of Massachusetts Lowell, USA

As autonomous robots continue to be deployed near people, robots need to be able to explain their actions. In this article, we focus on organizing and representing complex tasks in a way that makes them readily explainable. Many actions consist of sub-actions, each of which may have several sub-actions of their own, and the robot must be able to represent these complex actions before it can explain them. To generate explanations for robot behavior, we propose using Behavior Trees (BTs), which are a powerful and rich tool for robot task specification and execution. However, for BTs to be used for robot explanations, their free-form, static structure must be adapted. In this work, we add structure to previously free-form BTs by framing them as a set of semantic sets {goal, subgoals, steps, actions} and subsequently build explanation generation algorithms that answer questions seeking causal information about robot behavior. We make BTs less static with an algorithm that inserts a subgoal that satisfies all dependencies. We evaluate our BTs for robot explanation generation in two domains: a kitting task to assemble a gearbox, and a taxi simulation. Code for the behavior trees (in XML) and all the algorithms is available at github.com/uml-robotics/robot-explanation-BTs.

CCS Concepts: • **Computer systems organization** → **Robotics**;

Additional Key Words and Phrases: Behavior explanation, behavior trees, robot explanation generation, robot transparency, state summarization

1 INTRODUCTION

As robots are pushed by researchers and the industry to complete more complex tasks, improving the understanding of a robot's behaviors is becoming increasingly important. Prior work in **human-robot interaction (HRI)** has shown that improving understanding of a robot makes it more trustworthy [13] and more efficient [2]. To increase human understanding of robots, HRI

ACM Transactions on Human-Robot Interaction, Vol. 10, No. 3, Article 26. Publication date: July 2021.

26

researchers have mostly explored non-verbal physical behavior such as arm movement [16, 25] and eye gaze [34]. Such non-verbal behaviors can help people to anticipate a robot's actions [26], but do not provide insight into *why* the robot chose those actions. Direct explanations of why certain behaviors occurred can further improve one's prediction of behaviors [30]. Thus, robots need to explicitly explain their own behavior, similar to verbal human explanations.

Previously, we proposed a holistic blueprint of a robot explanation generation system consisting of three components: state summarization, data storage and querying, and a human interface [22]. The state summarization component is responsible for generating varying levels of summaries, either manually or automatically from different robot states, while performing tasks or from the stored states in introspection. In this article, we focus on the manual generation of high-level explanations for situations where robot programmers want to consider possible explanations to create more transparent robot systems as they program the functionality of the system.

We explore action sequence methods for task specification and execution as the foundation for robot explanation generation. Specifically, we base our work on *Behavior Trees* **(BTs)**, which is a tree structure that encapsulates behavior by control nodes that contain child execution nodes. BTs [12] are a popular way to model the behavior of AI agents in the gaming industry (e.g., References [27, 44]), but have also gained momentum in robotics, where they are used for end-user robot programming [40], deployed on a Rethink Robotics Sawyer robot [12], and applied to Learning from Demonstration [17] and navigation [29].

As we will discuss in Section 2.1, BTs have the advantages of modularity, reusability, and scalability over more traditional state machines. Behavior nodes, the basic structure of BTs, are both modular and reusable, which allows the system to scale while maintaining fewer unique modules [12]. In contrast, states in state machines are tightly coupled and do not scale well to more complex tasks, due to intertwined transitions. This is especially relevant for situations in which different tasks are interconnected and involve multiple steps.

Some of the disadvantages of BTs include the fact that they are free-form and static. As a counterexample of being unconstrained, robot behaviors can still be specified in a very deep tree—making the resulting BT hard to justify and natively unsupportive for generating hierarchical robot explanations of shallow depth, similar to their human counterparts. And being static, BTs lack the ability of dynamic modification (e.g., existing behavior insertion). This disadvantage limits us from inserting an explanation represented by a self-contained partial tree.

To solve these problems, we frame BTs in semantic sets with a goal structure so they are better suited for hierarchical robot explanation. We also propose explanation generation algorithms for the framed BTs, as well as an algorithm to create a self-contained behavior node that can then be dynamically inserted.

Additionally, we ground our framing structure and proposed algorithms with a multi-task, multi-step mobile manipulation kitting task. In this kitting task, a mobile manipulator robot assembles a kit of gearbox parts by navigating in a confined environment and collecting differently shaped, irregular parts (Figure 1) from different stations. It also involves a peg-in-hole large gear insertion task for machining a large gear. By first modeling this kitting task, consisting of many different subtasks, using BTs, we show the strength in execution and richness of expression of BTs, in addition to its simplicity and friendliness for non-roboticists, as argued in Reference [12] and shown by Reference [41]. We also demonstrate the application of our work to a non-manipulation task, the Taxi domain [15].

The primary contributions of this article are threefold:

(1) We first model a complex kitting task in BTs to show robot developers that BTs are capable of specifying complex high-level robotic tasks while not losing simplicity.
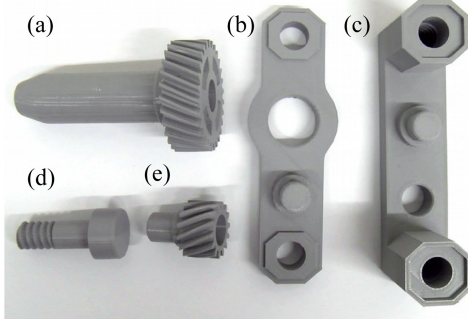
Fig. 1. Parts to be collected: (a) Large gear (b) Gearbox top (c) Gearbox bottom (d) Screw (e) Small gear. Note that the large gear (a) is meant to be machined to have threads; the large gear must be inserted into a machine for this process to occur.
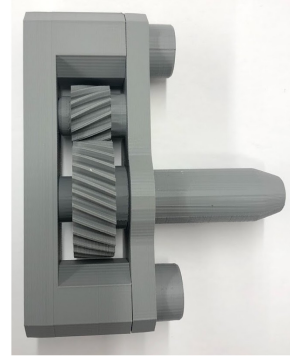
Fig. 2. Assembled gearbox using the required mechanical parts that the robot placed into the caddy and delivered to the inspection table.

(2) Given that BTs can be deep, we frame BTs into semantic sets and contribute algorithms to generate hierarchical explanations, taking the node types in BTs into account.

(3) We make the static BTs dynamic so self-contained behavior nodes can be inserted dynamically as a subgoal, as a member of the semantic set.

To validate the framing structure and proposed algorithms for explanation generation, we evaluate them on two diverse tasks, the first involving robot manipulation and the second involving navigation planning. The first task includes the large gear insertion subtask of the kitting task, in addition to the screw picking and placing subtasks that we used throughout this work to give readers a concrete idea about our work in the early phase. The second task involves a taxi routing problem that attempts to optimize an agent's navigation path while completing subtasks such as picking up and dropping off a passenger. By evaluating our explanation generation algorithms in these two different tasks, we demonstrate the algorithms' generalization to domains with differing complexities and actions.

To the best of our knowledge, this is the first work on using behavior trees for explanation generation. We conclude with a discussion of this work, including its limitations and ideas for potential future work.

2 RELATED WORK

2.1 Robot Task Representation: Why Behavior Trees for Robot Explanations

For a robot to better perform tasks for which only humans are typically adept, robust task representations are important [35]. It is critical to have a readable and user-friendly representation for robot explanations to minimize the mental burden of mapping robot task representation and execution to explanations. After perception, robot tasks can be decomposed to actions and action sequences for execution [20]. Nakawala et al. [35] listed different methods for robot action sequence presentations and discussed a few popular approaches including ontology, state machines, and Petri Nets. We will discuss the merits and drawbacks of each of these approaches and why behavior trees might be a better solution for robot programmers to manually provide robot explanations.

Ontology belongs to the knowledge representation family; this approach attempts to infer the task specification from high-level, abstract, underspecified input from a predefined set of actions, e.g., put the cup left of the plate. Two popular ontology implementations are KnowRob [4, 49] and CRAM [5], which uses KnowRob. While CRAM[1] uses the Common Lisp language, KnowRob embeds a number of common actions observed by leveraging the Prolog language and its logical predicates. As our previous work [22] discussed, KnowRob and CRAM introduce another programming paradigm, logical programming, to the robot system; this paradigm is distinctly different from C++ and Python used in the popular **Robot Operating System (ROS)** [43] (i.e., the procedural and object-oriented paradigms). Additionally, the knowledge representation approach does not examine how to specify tasks [41], leaving important details out of users' control.

State machines have a long history in Computer Science [45] and in robotics [38]. Unlike the traditional notion of a state, which is conventionally a world state, a state here is an execution in the task flow. A notable **finite state machine (FSM)** implementation is the **SMACH ("State MACHine")** library [7], which is tightly integrated with ROS. SMACH offers two interfaces: State and Container. A State represents an execution with a set of outcomes, while a Container is a policy composed of a set of states, which can be used to encapsulate behavior. SMACH allows the hierarchical composition of containers (i.e., state machines), which are also states with outcomes. States are transitioned through outcomes. Researchers have used SMACH to represent and execute tasks such as serving drinks [8] and more general household tasks [36]. RAFCON [10] is another implementation of a hierarchical FSM, very similar to SMACH but with a well-designed graphical editor, including the ability to visualize deep hierarchies by panning and zooming. RAFCON has been demonstrated to specify complex tasks such as planetary exploration [9, 47].

While a state machine can describe taskset workflow and is very flexible and well-known, there are a few notable disadvantages. Colledanchise and Ögren [12] describe maintainability, scalability, and reusability issues. A workflow represented by a large number of states or hierarchies is hard to maintain, scale, and is prone to design errors: Adding a new state requires careful examinations [12] of incoming and outgoing transition dependencies, where the tight coupling of states makes it hard to reuse certain transition-states. For robotics specifically, it adds unnecessary complexity especially for simple linear tasks without loops, such as a primitive pick manipulation task. In addition, a manipulation state machine breaks the fluency of such manipulation primitives, generating discrete arm movement that is paused, thus disconnected instead of a smooth multi-waypoint trajectory. Also, as Nakawala et al. [35] point out, state machine implementations are code-intensive and challenging for complex tasksets. For a comprehensive discussion of BTs and state machines, we refer readers to the Behavior Trees book by Colledanchise and Ögren [12].

Peri Nets (PNs) and **Behavior Trees (BTs)** are of growing interest. They share the same theoretical concepts of state machines and have equivalent state machine representations. Peri Nets, with sharing of tokens between states, are designed to be capable of modeling concurrency and distributed execution and coordination, with applications in multi-robot systems [52] and soccer [39]. However, concurrency can be achieved using framework-agnostic programming language constructs that are well-known across programming languages. Distributed execution can be achieved by using distributed frameworks like ROS or more general frameworks such as Apache Spark[2] for big data. Nonetheless, our work does not focus on concurrency and distributed execution and coordination.

Compared to state machines, **BTs** have some key advantages: modularity and reusability by behavior unit, expressiveness, and human readability by coherent and compact structure units [12].

---

[1]https://github.com/cram2/cram.
[2]https://spark.apache.org/.

Among state machines, Peri Nets, and BTs, only the behavior tree approach is inherently user-friendly through human readability—the tree representation is simple, sharing familiar terminology of behaviors and using the analogy of a physical tree, removing the extra layer of abstraction of states and transitions, or operators and tokens. This allows for less information loss while we convert the underlying representation to explanations. In the past decade, BTs have been used for pick-and-place tasks [3] and end-user programming to instruct industrial tasks using a UR5 robot arm such as wire bending and sanding [19, 40]. Results of a user study on a BTs-based robot programming interface [41] indicate that BTs are a practical and effective representation for specifying robot programs. Again, for a comprehensive introduction, we refer readers to the Behavior Trees book published by Colledanchise and Ögren [12].

Thus, we choose BTs for hierarchical robot explanation generation, especially high-level tasks. Throughout this work, we also use BTs for primitive manipulation tasks after motion planning for simplicity. However, in Section 8, we also suggest using **MoveIt task constructor (MTC)** [18] for connecting sementicless waypoints in those manipulation primitive tasks, to not only achieve smooth multi-waypoint arm trajectory, but also to give us more information about the black-box probabilistic motion planning process, which provides potential for low-level explanations in the future.

2.2 Tree-based State Summarization

Researchers have been studying how to generate varying levels of summaries either manually or automatically. A common approach is for developers to manually create categories by which the robot can explain its actions, for example, programmer-specified function annotations for each designated robot action are used in Reference [23]. By creating a set of robot actions, correlated with code functions, the system is able to snapshot the state of the robot before and after a function is called. Since the state of the robot could be exceedingly large in a real-world system, the state space is shrunk by isolating which variables are predetermined to be most relevant. These annotated variables are recorded every time a pre- and post-action snapshot is made. The robot then uses inspection to compare the pre- and post-variables of one action, compared to other similar successful actions, to make judgments. Tree structures have been explored but not used as a robot task specification and execution tool. A discussion of additional state summarization work can be found in Reference [22].

Under the umbrella of shared plan execution, Devin and Alami [14] described a Theory of Mind system to estimate the human partner's mental states but did not explore how the robot would express the assessment. In a shared pie-cooking preparation scenario task, Milliez et al. [33] proposed to use a binary tree-based system for human-robot collaborative planning adaption for different human knowledge levels (i.e., new, beginner, intermediate, and expert). As explaining the plan was not the focus, the work concentrated on determining whether to explain based on the human knowledge level. The robot would explain every step to new collaborators, would ask beginners if they wanted an explanation, simply tell the current step to intermediate users, and offer no explanation or prompt at all for expert users.

Kaptein et al. [24] adapted hierarchical task analysis [46] to the **Goal Hierarchy Tree (GHT)**. This involves creating a tree where the top node is a high-level task, which can be broken into a number of sub-goals, each linked by a belief (i.e., condition). Each sub-goal can then be broken into either sub-goals or actions. Choosing one sub-goal or action over another is based on a belief. The GHT is then used to generate explanations by returning the description of the nodes. When comparing goal-based vs. belief-based explanations, results found adults, compared to children, prefer goal-based rather than belief-based explanations. However, this work focused on cognitive reasoning rather than behavior explanation or behavior programming

and execution. While a NAO robot was used, the work did not involve any physicality such as arm or leg movement. In contrast, our work builds the foundation to provide a complete robot task specification and execution solution for robot developers, especially in the manipulation domain, who ultimately generate explanations either manually or by proposing state summarization algorithms.

3 BACKGROUND: FORMULATION OF BEHAVIOR TREES

Behavior Trees have previously been used for gaming agents but have gained momentum in the robotics community during the past few years [11, 12, 19, 32, 42]. In this section, we borrow from the previous work to formulate Behavior Trees using a simplified yet complete set of notations.

A behavior tree $T = \{C, E, Edges\}$ is an ordered rooted tree where *control flow nodes*

$$C = \{Sequence, Fallback, Parallel, Decorator\}$$

are internal or non-leaf nodes that have children nodes, and *execution nodes* $E = \{Action, Condition\}$ are external or leaf nodes. Control flow nodes can have control flow or execution nodes as their children, whereas execution nodes are execution units that do not have any children.

Execution starts or *ticks* from the root node of the tree. Control flow nodes route the ticks to its children until the execution nodes returns a status $s \in S = \{$ RUNNING, SUCCESS, FAILURE $\}$. The status of a control flow node depends on one of its descendant nodes.

A sequence control flow node, indicated by "→", executes each of its child nodes sequentially. The sequence returns SUCCESS only if all of its child nodes return SUCCESS and returns RUNNING or FAILURE if any of its children returns such status.

A fallback control flow node, denoted by "?", also executes its children sequentially but only requires one child node to succeed to return SUCCESS. If a child node returns FAILURE, then the fallback node will tick its next child until SUCCESS or FAILURE. If all of its children return FAILURE, then the fallback node returns FAILURE. Similar to the sequence node, it returns RUNNING if any of its children returns such status. The fallback node type is also known as a selector.

A parallel control flow node, labeled as "⇉", executes all of its children simultaneously or the children are all asynchronous themselves. Otherwise, the parallel node has the same control logic as a sequence node. Callers can specify thresholds to make the node return SUCCESS or FAILURE when a certain number of children nodes returns one of the statuses.

A decorator control flow node, often represented with a diamond shape, can only have one child. It applies a function f to its child to manipulate the returned status. Examples include an *inverter* node that negates its child's status and a *retry* node that ticks its child N times as long as a FAILURE status is returned.

Usually boxed, an action node is an execution unit that runs a piece of code. It returns SUCCESS upon successful completion and FAILURE if it is impossible to complete. RUNNING is returned during execution.

Often circled, a condition execution node checks whether a state is met, making it a special case of the action node. However, it does not return the RUNNING state; otherwise, it functions the same as an action node.

Table 1 lists all the nodes and their returned states.

4 MODELING ROBOTIC TASKS USING BEHAVIOR TREES

Even though the BT method is relatively simple, which makes it easy to understand for non-roboticists as previously mentioned [41], it does not mean that BTs are incapable of representing

Table 1. Classical Behavior Tree (BT) Formulation: Nodes and Return Statuses

Notation	Node	Node Type	$s =$ SUCCESS	$s =$ FAILURE	$s =$ RUNNING
$\rightarrow$	*Sequence*		All children $\rightarrow s$	Any child $\rightarrow s$	Any child $\rightarrow s$
?	*Fallback*	Control	Any child $\rightarrow s$	All $\rightarrow s$	Any child $\rightarrow s$
$\Rightarrow$	*Parallel*		M children $\rightarrow s$	$N - M$ children $\rightarrow s$	Any child $\rightarrow s$
Diamond	*Decorator*		Function $f \rightarrow s$	Function $f \rightarrow s$	Function $f \rightarrow s$
Boxed	*Action*	Execution	Self $\rightarrow s$	Self $\rightarrow s$	Ticking
Circled	*Condition*		Self $\rightarrow s$	Self $\rightarrow s$	-

(Adapted from [12]).

complex tasks or to execute those tasks. To demonstrate the execution model and the expressiveness of BTs, we now describe a real-world scenario of a complex gearbox kitting task and how one can represent tasks in BTs. We will refer this task when we describe how we use BTs for robot explanation.

4.1 The Gearbox Kitting Task

The gearbox kitting task is a challenging task designed for the 2019 FetchIt! Mobile Manipulation Challenge,[3] organized by Fetch Robotics and held at the IEEE Conference on Robotics and Automation (ICRA) in May 2019. In this competition, a Fetch mobile manipulator robot [50], which has a single chest-mounted 7 DoF arm with a torso lift joint and a head-mounted RGBD camera, needed to be programmed to assemble as many kits as possible. Kit assembly is achieved by navigating to different stations to pick a specified number of complex, irregular mechanical parts, including screws, gears, and gearbox container parts (Figure 1), placing them into specific compartments of a caddy, and finally picking the full caddy up and delivering it to the inspection table for final gearbox assembly by factory or warehouse workers. The robot needs to first collect two screws from a bin as well as one each of a large gear, small gear, gearbox bottom, and gearbox top piece. We described our efforts to address the challenges present in this task in Reference [21].

While the robot can complete the kitting task alone, the ultimate goal is to apply it to a collaborative scenario where one or two workers stay at the inspection table to inspect and perform the fine motor skills to assemble a functional gearbox (Figure 2) from the parts that the robot collected, requiring human-robot collaboration.

This human-robot collaboration scenario provides the tasks and the environment for a reasonable testbed. Because there are several opportunities for unexpected or opaque events to happen, where a human may want to ask for a robot to explain itself.

For example, a common occurrence is that the Fetch robot may not be able to grasp a caddy or a gearbox part if it is placed too close to a wall. Fetch's arm may not be long enough to reach given the constraints presented by the end-effector orientation (it must be pointed down to grasp the caddy) and standoff distances imposed by the dimensions of the tables. This scenario is not apparent to novice users or bystanders who do not have intimate knowledge of Fetch's characteristics. Even for roboticists, forming an inaccurate mental model of the robot from time to time is still possible. In such scenarios, Fetch should initiate an explanation to inform the user.

---

[3]https://opensource.fetchrobotics.com/competition.

Table 2. Notation of and Changes to Behavior Trees as Used
in This Article

Symbol	Color	Node	$s =$ RUNNING
品	White	*Subtree*	**Child** $\rightarrow s$
$\rightarrow$	Pink	*Sequence*	Any child $\rightarrow s$
?	Blue	*Fallback*	Any child $\rightarrow s$
	Varies	*Decorator*	Function $f \rightarrow s$
	White	*Action (A)*	Ticking
	Green	*Condition (C)*	**Ticking**

Another common occurrence is confusion when differentiating between two gearbox parts that appear similar in height via point cloud due to sensor noise. This can make the object detection fail, causing the robot to grasp the incorrect object. In this scenario, a human might initiate a robot explanation, as the robot may not be aware that it performed incorrectly.

Finally, a human-initiated robot explanation might also be needed when the robot stops at a different location in front of the caddy table than what was expected, and places a part into the incorrect caddy. This could occur due to navigation error range and the narrow horizontal field of view (54°) of its RGBD camera, which may cause part of the caddy to be occluded.

While scoped for a manufacturing environment, the same types of tasks are relevant to home environments (e.g., navigating between areas in a narrow hallway kitchen and a dining table, and manipulating objects in these places).

In this article, we focus on human-initiated high-level robot explanation.

4.2 Revised Notation for Behavior Trees

To visualize Behavior Trees, we adopted the Groot software[4] and its notations. In contrast to the classical BTs notations summarized in Table 1, a subtree node is added. Every node is boxed to accommodate more text descriptions instead of a single word.

Text color is used to differentiate node type and custom decorator nodes on a dark gray background. For control nodes except for decorators, pink and blue indicate sequence and fallback, respectively. For execution nodes, white and green indicate action (A) and condition (C) nodes. Because a decorator node is attached with a custom function f, they have different colors.

The subtree node can only have one child, similar to a decorator node, and is denoted by a tree hierarchy icon. Indeed, it can be deemed as a special case of a decorator with the function f simply doing nothing but ticking its child.

Table 2 shows the revised notations including the new subtree node and the color codes.

4.3 Modeling Using Behavior Trees

Figure 4 shows the top level of the gearbox kitting task represented as a BT where each subtask is encapsulated in a subtree. To improve the readability of the tree in this article, all subtasks were left collapsed in the figure.

Figures 5 and 6 illustrate the specification of two representative subtasks in the kitting task using BTs: picking a screw placed in a bin on the table and placing a screw into a specific

---

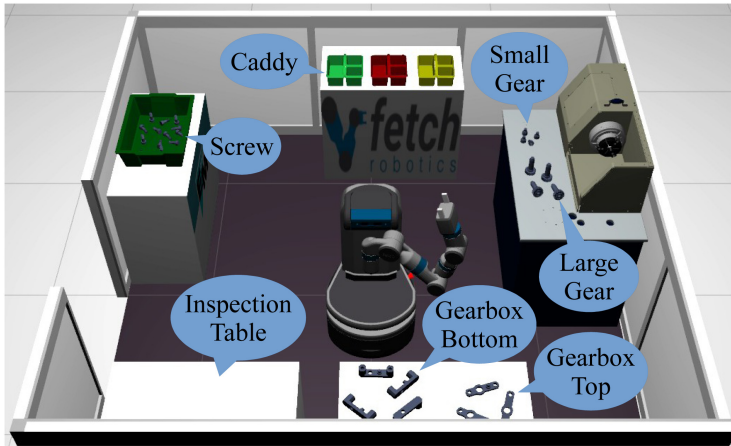[4]https://github.com/BehaviorTree/Groot/.

Fig. 3. The arena where the gearbox kitting task is carried out by a Fetch robot. Rendered in Gazebo, the main goal is to place a specified set of parts into the correct sections of the caddy, then to transport the caddy to the inspection table.

caddy compartment (see Figure 3 left and top). Other tasks in the kitting task are similar and can be represented in the same way, thus highlighting the modularity and reusability of BTs by reusing existing tree nodes. For example, picking gearbox tops/bottoms and large/small gears are similar to screw picking. Placing those parts into a caddy is the same as placing a screw into the caddy except that the compartment is different. If you compare Figures 5 and 6, more than half of the nodes are reused. However, machining a large gear is a unique task. We visualize it in Figure 9 and will discuss in Section 8 during evaluation in hope of generalization. The examples demonstrate every type of node in both the control and execution categories except for the parallel node. "RetryUntilSuccesful" and "ForceFailure" are two types of decorator nodes.

As illustrated in Figure 5 and 6 as well as Figure 9, each node is visualized with its node ID on the first line, node name on the second line, and input and output (I/O) ports following. Custom nodes can register their own IDs and are mostly at the leaf level (i.e., action and condition nodes). Node names are not needed for registration and only used when the node is reused to clarify the robot behavior it represents.

With the built-in nodes listed in Table 2 and registered custom nodes, the whole tree, including descriptive names and I/O ports, is specified in an XML file that is then parsed for execution or visualization. As an example, the bottom right action node in Figure 5, `retreat arm`, has the ID of "retreat arm" and an empty name, because the ID shows the specific behavior that the node represents. At the same level, the "reach" node is reused three times and assigned a name each time to differentiate each other. As we will see later during explanation generation, the node name plays an important role, as it can represent the robot behavior.

Input and output ports are the light gray fields filled with string values. Similar to the previously mentioned SMACH [7], these ports are designed to be data-driven and passed around between nodes to share data. Input ports can be specified by a literal value or a dynamic key, while output ports, named "output," can only be keyed. For example, the top right action node in Figure 5, "`convert to grasp pose`," has a "`from`" input port with a "`{object pose}`" key and a output port with a "`{grap pose}`" key. Each tree or subtree maintains a key-value dictionary to store outputs resulted from some nodes and to be used by other nodes in the tree or subtree. If

subtree nodes are used, then each dictionary is scoped within the subtree to avoid naming space collision for the keys and be easy to reason about.

4.4 The Screw Picking Subtask: Tree Breakdown

As visualized in Figure 5, the go pick screw task starts as a fallback node (depth 2), which ticks the sequence node with the same name. If the sequence node fails, then the ask node will be executed to ask humans to intervene. In the children of the sequence node (depth 4 and 5), a robot will first move to screw station and, if unsuccessful, retry three times. Then the robot will detect screw and pick screw, and retry up to three times until successful. A sequence node with an empty name (depth 5) is needed, as the RetryUntilSuccessful node is a decorator node that can only have one child. To detect screw, the robot needs to lift torso, look at table, and detect screw, which outputs object pose of the screw. A fallback node is used to pick screw (depth 6) so, when the sequence of pick screw failed, the robot will reset arm (depth 7 bottom) before retrying (the second child at depth 4). To pick screw, the robot will run all execution nodes at depth 8. For the action node of allow partial collision, it will crop a 4 mm cube at object pose with 3 mm padding from the collision scene, allowing the gripper to collide with the detected screw.

The go place screw task in Figure 6 can be broken down in a similar fashion. Note that this screw placing task shared a considerable amount of sequence and action nodes, illustrating the modularity and reusability advantages of using BTs again. Thanks to this, once the first behavior tree is coded, similar tasks can be easily encoded in behavior trees using existing nodes.

5 FRAMING BEHAVIOR TREES FOR HIERARCHICAL EXPLANATION GENERATION

Behavior Trees are a free-form action sequence specification and execution tool that only provide a preset of control nodes with simple control flow algorithms, imposing a minimum amount of structural rules. This structure provides ultimate flexibility for robot or AI developers to program behaviors for different application needs or use cases, such as the manipulation tasks in this article.

However, to provide concise, multi-level explanations [22], we propose to frame BTs for hierarchical explanation generation, given the experience from implementing the subtasks in the kitting task with BTs.

The behavior tree of a multi-task, multi-step task can be simplified and decomposed into a set of semantic sets:

$$\{thegoal, subgoals, steps, actions\}.$$

To give a concrete understanding, now we examine the kitting task tree. As shown in Figure 4, the *goal* is to build a gearbox kit and its subgoals are all of the leaf nodes. A *subgoal* can be further broken down into *steps*, which can be nested to span more than one level or as shallow as a flat one-level tree. Finally, the ending steps consist of a set of *actions* (i.e., execution nodes).

Figures 7 and 8 show the framed behavior trees of the screw picking and placing subtasks. The framed tree can be generated by ignoring decorator nodes and the control nodes that have a child sharing the same name. For example, all RetryUntilSuccessful decorator nodes are ignored, and the fallback nodes at depth 2 and 6 are also ignored (see Figure 5). Note that the move to node is a special case of a step, which reveals that a step does not have to be a control node.

Compared to the original behavior trees in Figures 5 and 6, the hierarchy is much simplified and more clear, making it more suitable for explanation generation. Section 6.2 details the generation algorithms that used the framed trees under the hood.
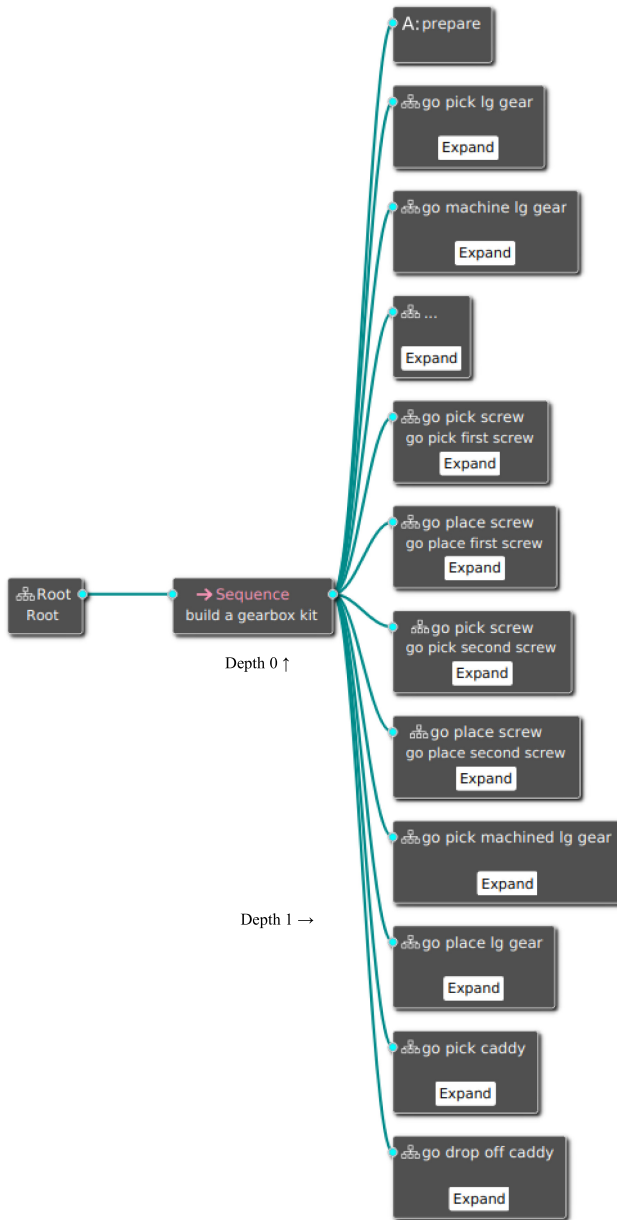
Fig. 4. The top level of the gearbox kitting task represented as a sequence of subtrees in a Behavior Tree. For readability, six tasks—go {pick | place} {small gear | gearbox top | gearbox bottom}—are represented by "…". Note that the root node on the top is merely a pointer to the real root node in the middle, which is why depth 0 is at the sequence node.

6 ALGORITHMS ON BEHAVIOR TREES FOR ROBOT EXPLANATION GENERATION

Behavior Trees are a static action sequence method. Once created, it does not allow for the dynamic addition of behavior nodes. BTs are also not designed to be interactive. The behavior tree associated with a task might be presented on a display screen for introspection, but when an end-user asks
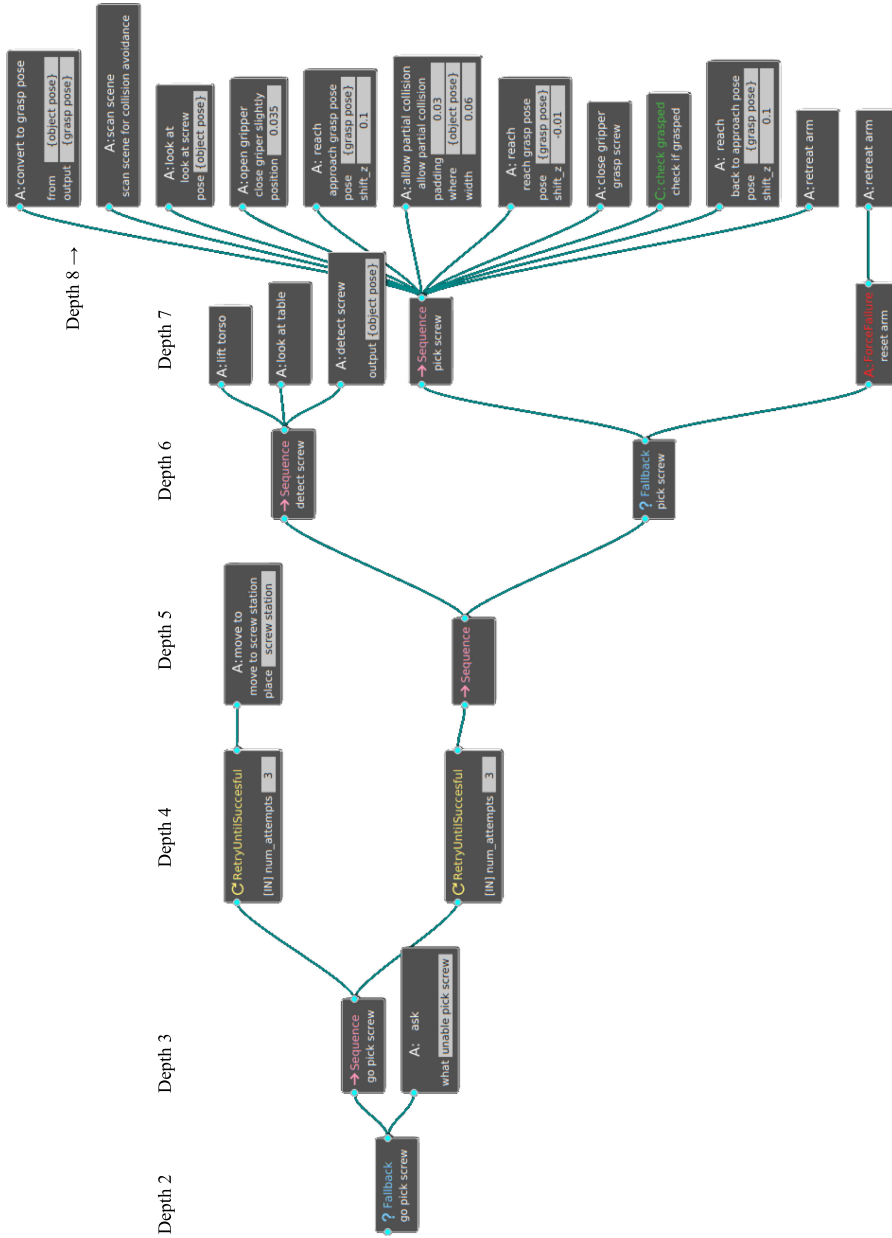
Fig. 5. The representative screw picking subtask modeled in Behavior Trees. The leftmost dot indicates that the fallback node has a parent, but the subtree parent node and other ancestor nodes are hidden here, as they are shown in the previous figure.

Fig. 6.   Screw placing, another representative subtask of the gearbox kitting task, modeled as a Behavior Tree. Similar to the previous figure, the leftmost node's ancestors are not shown. For more detail, refer to Section 5.

Fig. 7. Simplified, semantic sets for the screw picking subtask. The goal is not shown here, as Figure 4 is sufficient without any simplification.

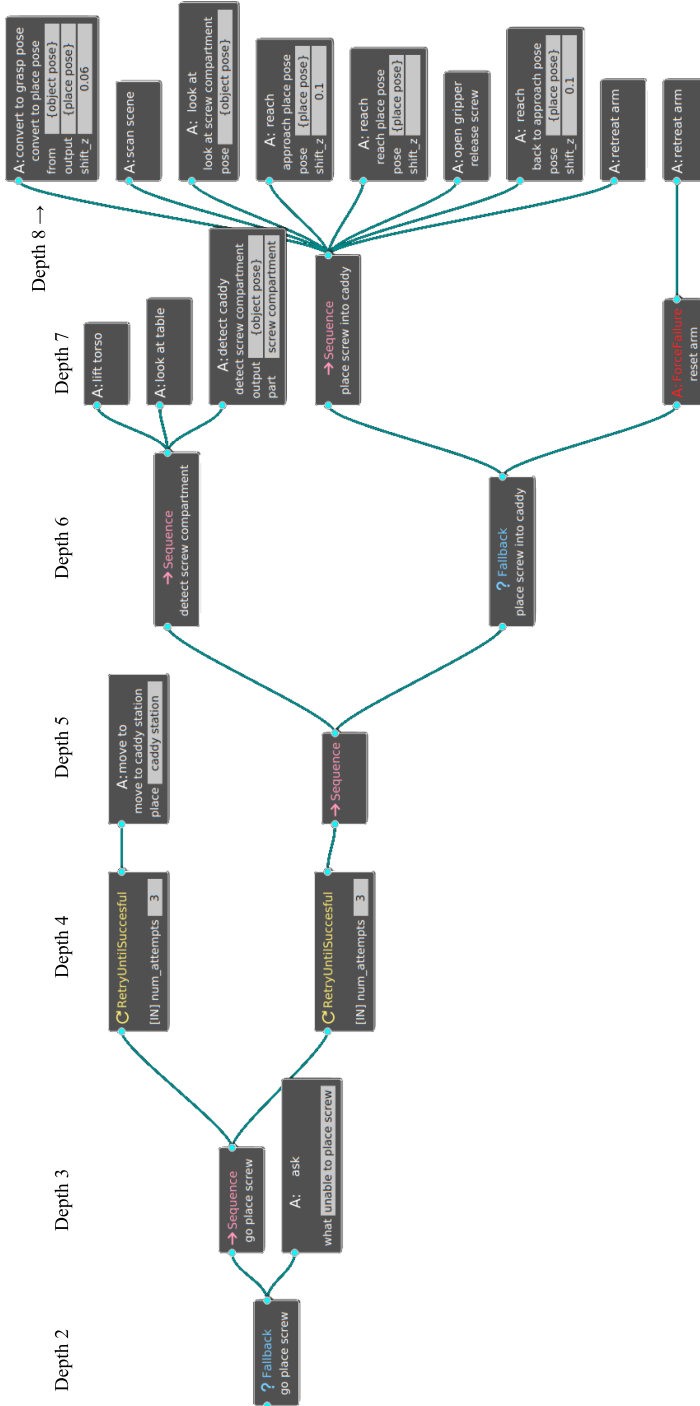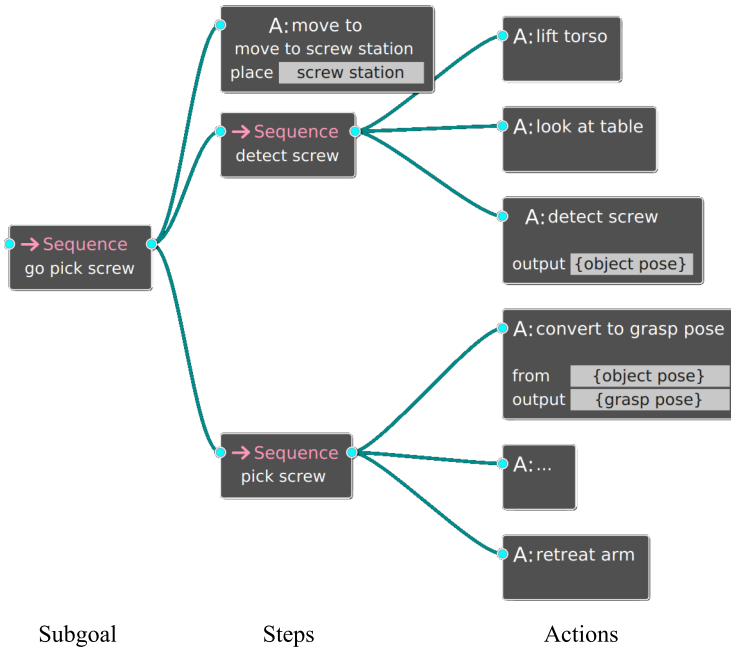what part of the task the robot is attempting to finish or why the robot is doing its current sub-task, it is unknown how to query at different nodes or what should the robot reply given different functions of different types of node. To solve this issue, we propose the following algorithms. All the pseudo-code is derived from our C++ implementation using the `BehaviorTree.CPP` library.[5] The C++ implementation is available at https://github.com/uml-robotics/robot-explanation-BTs.

ALGORITHM 1: Answer "What are you doing?" (Q1)

    Input: Node n (Current node in execution)
    Output: String *answer*

1 *// p.short_description ← **if** p.has_name **then** p.name **else** p.ID **end**;*
2 **return** *"I" + p.short_description + "."*;

6.1 Supporting Querying Current State

Because any BT node returns a status $s \in S = \{$ RUNNING, SUCCESS, FAILURE $\}$ and condition nodes do not return a RUNNING state, we reintroduce the RUNNING state back to condition nodes to know which node is currently running. This change is reflected in Table 2.

This change is necessary, because robots operate in the real three-dimensional world in which completing a checking task is more time-consuming than checking a condition in a virtual world. Behavior Trees are popular in the gaming industry, in which inputs are accurate, checks are fast, and noisy sensors do not exist. However, it may take time for an embodied robot to check if a state is reached (i.e., execute the condition node). For example, to check if a screw in a bin is

---
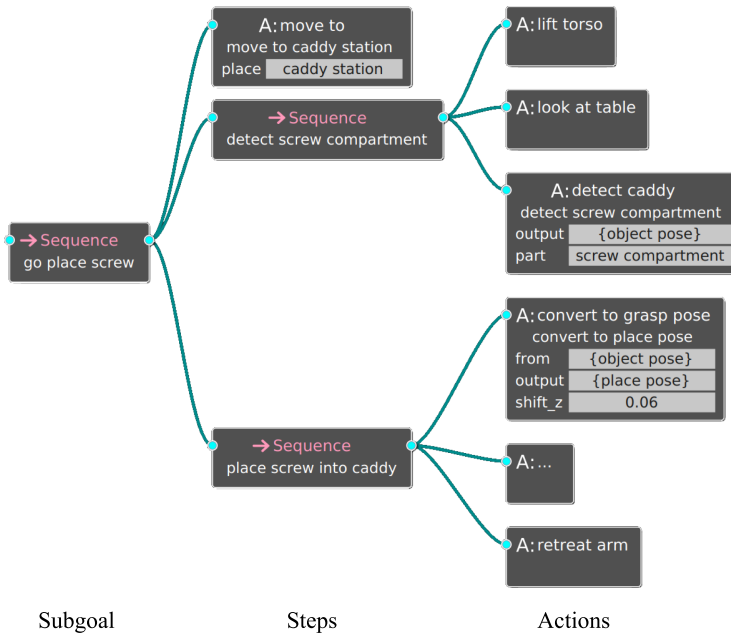
[5]https://www.behaviortree.dev/.

Fig. 8. Simplified, semantic sets for the screw placing subtask.

Table 3. Questions to Be Answered during Hierarchical Explanation

	Question	Algorithm
Q2	Why are you doing this?	Algorithm 2
Q3	What is your subgoal?	Algorithm 3
Q4	What is your goal?	Algorithm 4
Q5	How do you achieve your { goal | subgoal }?	Algorithms 5, 6

reachable, it can take a while for sampling-based motion planning methods to avoid collisions for small object manipulation, especially when the collision objects are further away and the objects are represented as a number of small units, making the scene representation time-consuming to search.

Given that every execution node can be in the state of RUNNING, we can use the concept of a state listener to track which node is ticking. The tracked node with the RUNNING state can then be used to answer the most basic question: Q1. "What are you doing?" For the sake of showing all of the algorithms in this text, we include this as Algorithm 1.

6.2 Supporting Hierarchical Explanation Generation

Once we have the framed Behavior Trees for hierarchical robot explanation, the context to answer the questions listed in Table 3 is established by the sets of semantic sets, in addition to Q1.

Compared to Q1, these questions are chosen because humans tend to seek causal knowledge from explanations [6, 28]. Q2 gives the causal information of Q1. Q3 gives the intermediate cause, while Q4 offers the final cause—"the end, function or goal" [28] and Q5 is designed in the hope of providing detailed steps to improve understanding. Q1–Q4 are about thinking backward in terms of the tree structure, while Q5 is forward-thinking, from left to right.

ALGORITHM 2: Answer "Why are you doing this?" (Q2)

> **Input**: Node *n* (Current node in execution)
> **Output**: String *answer*

1   *p* ← *n*.parent;
2   // *find a non-decorator ancestor of "n" that has a name different from "n"*;
3   **while** *p* ≠ *null* **or** *p.type = Decorator* **or** *p.has_name()* **or** *p.short_description = n.short_description* **do**
4 | *p* ← *p*.parent;
5   **end**

6   **return** *"I" + p.short_description + "in order to" + p.name + "."*;

ALGORITHM 3: Answer "What is your subgoal?" (Q3)

> **Input**: Node *n* (Current node in execution)
> **Output**: String *answer*

1   *p* ← *n*.parent;
2   **while** *p* ≠ *null* **and** *p.type* ≠ *Subtree* **do**
3 | *p* ← *p*.parent;
4   **end**

5   **if** *p* ≠ *null* **then return** *"My subgoal is to" + p.name + "."*;
6   **else return** *"Sorry. I don't have a subgoal."*;

ALGORITHM 4: Answer "What is your goal?" (Q4)

> **Input**: Node *root*
> **Output**: String *answer*

1   **return** *"My overall goal is to" + root.name + "."*;

Note that we do not focus on speech recognition or **natural language processing (NLP)**. With the underlying algorithms described in this section, one could plug in an NLP framework to extract the semantic information from the question. In our work, we are also not concerned with the grammatical correctness of the explanations; we could also add additional language generation capabilities.

Algorithm 2 shows the steps to answer Q2, reasoning about the current behavior. Lines 1–5 find the non-decorator control ancestor node *p* of the node in execution *n* so *p* has a name and the name is different from *n*'s. Note that the short description of a node is described in Algorithm 1: It is the name of the node *n'* if *n'* has a name, otherwise the ID of *n'*.

Algorithm 3 shows the steps to answer Q3 about subgoal. Lines 1–4 find the subtree ancestor node of *n* as the subgoal.

To answer Q4, the goal is simply the name of the root node, as indicated by Algorithm 4.

To answer Q5, Algorithm 5 shows the steps where a depth-first search is performed at the goal or subgoal node *g*. In lines 3–6, a descendant node is added to the set of steps if the node being traversed is not a decorator node, has a name, and does not have the same name as *g*. Similar to previous algorithms, Algorithm 6 shows the generated explanation text.

6.3 Supporting Failure Explanation Generation

In addition to asking for explanations for the current behavior, explanations from failure handling are of interest. For example, "Was there anything wrong?," "What went wrong?," and "How was the failure handled?"

ALGORITHM 5: Find Steps From a Goal or Subgoal Node

    Input: Node g (The goal or subgoal node)
    Output: Set *steps* (An ordered set of nodes)
1   *steps* $\leftarrow \emptyset$
2   DepthFirstSearch : g, f(node) $\rightarrow$ **begin**
3   **if** *node.has_name()* **and** *node.name* $\neq$ *p.name* **and** *node.type* $\neq$ *Decorator* **then**
4   *steps* $\leftarrow$ *steps* $\cup$ {*node*};
5   do not traverse further;
6   **end**
7   **end**
8   **return** *steps*;

ALGORITHM 6: Answer "How do you achieve your { goal | subgoal } ?" (Q5)

    Input: Node n (Current node in execution)
    Input: Node g (The goal or subgoal node found in Algorithm 3)
    Output: String *answer*
1   **if** g = *null* **then**   // *specific to subgoal*
2   **return** *"Sorry. I don't have a subgoal."*
3   **else**
4   *steps* $\leftarrow$ output of Algorithm 5 given g;
5   **return** *"To achieve the {goal | subgoal}"* + *g.name()* + *", I need to"* + *steps.to_string()*;
6   **end**

All those questions can be answered by Algorithm 7, which centers around Fallback and RetryUntilSuccessful nodes that handle failures. For a Fallback node, when its status is "SUCCESS," we can check whether the first children nodes in its sequence were unsuccessful (line 4), i.e., to see whether the execution fell back to other child nodes. Those children nodes with "FAILURE" status (lines 7–9) can then be used for explanations of failures, more specifically, what went wrong (lines 6 and 10–12). For example, in the Fallback node at depth 6 of Figure 5, if its "pick screw" child node failed because the "check if grasped" failed, the robot can say "I could not go pick screw because check if grasped failed." Potentially, it can be further explained by attaching information such as perception results and motion trajectories; the latter two can be presented on a monitor or by a projector.

For a RetryUntilSuccessful decorator node, it can also be used to answer the questions, especially that the failure is being handled by retries. The relevant code in Algorithm 7 are lines 13–33. Lines 13–23 handle the case when the Retry node has started its first attempt (line 13), indicating one node has failed. Then, we can find its parent to explain what is being attempted (lines 15–17). Similar to the Fallback node, the failed node is then found to explain what went wrong (lines 18–22). Lines 25–33 consider the case where a Fallback node has an ancestor Retry node. In this case, we want to inform the information at the retry node (which attempt and what it is attempting to do) in addition to the Fallback information in the previous paragraph. Note that the algorithm does not handle the case that a Retry node has an ancestor Fallback node. From our experience, we found that Fallback is a better choice to handle failure than Retry, because there might be another preferred method over retrying.

6.4 Supporting Dynamic Behavior Insertion as Subgoal

Behavior Trees have the advantages of modularity and reusability, but we cannot simply take any node n' and insert n' after the current execution node n, because some behavior nodes

---

ALGORITHM 7: Answer "Was there anything wrong?" "What went wrong?" "How was the failure handled?"

    Input: Node n (Current node in execution)
    Output: String $answer$

1 $p \leftarrow n$.parent;
2 $is\_wrong, fell\_back \leftarrow$ **false**;
3 **while** $p \neq null$ **and** $p.type \neq Subtree$ **and** $is\_wrong \neq$ **true do**
4   **if** $p.type = Fallback$ **and** $p.children[0].failed$ **then** // Fallback node
5        $is\_wrong, fell\_back \leftarrow$ **true**;
6        $answer \leftarrow$ "I could not" + p.short_description() + "because";
7        $n_{fail} \leftarrow$ DepthFirstSearch : p, f(node) $\rightarrow$ **begin**
8   **if** $node.failed$ **and** $node.type \in \{Condition, Action\}$ **then return** $node$;
9   **end**
10   **if** $n_{fail}.parent \neq null$ **and** $n_{fail}.parent.short\_description = p.short\_description$ **then**
11         $answer$ += "I was unable to" + n_{fail}.parent.short_description + "as";
12       $answer$ += n_{fail}.short_description + "failed.";
13   **else if** $p.type = Retry$ **and** $p.attempt > 0$ **then** // Retry node
14        $is\_wrong \leftarrow$ **true**;
15        $rp = p$.find_non_null_parent;
16   **if** $rp \neq null$ **then**
17          $answer$ = "I am retrying for attempt" + p.attemp + "to" + rp.short_desc + ".";
18          $n_{fail} \leftarrow$ DepthFirstSearch : p, f(node) $\rightarrow$ **begin**
19   **if** $node.failed$ **and** $node.type \in \{Condition, Action\}$ **then return** $node$;
20   **end**
21          $fp = n_{fail}$.first_ancestor_with_name;
22          $answer$ += "I could not" + fp.short_desc + "because" + n_{fail}.short_desc + "failed."
23    $p \leftarrow p$.parent
24 **end**
25 **if** $fell\_back$ **then** // Check if the Fallback node has Retry parent
26    $p \leftarrow fallback\_node$.parent;
27   **while** $p \neq null$ **and** $p.type \neq Subtree$ **do**
28   **if** $p.type = Retry$ **and** $p.attempt > 0$ **then**
29        $rp = p$.find_non_null_parent;
30   **if** $rp \neq null$ **then**
31   a += "I am retrying for attempt" + p.attempt + "to" + rp.short_desc + "."
32      $p = p$->getParent();
33   **end**
34 **if** $not(is\_wrong)$ **then** $answer$ = "Nothing went wrong.";
35 **return** $answer$;

---

have dynamic input ports and are thus dependent on some other behavior nodes providing corresponding output ports. For example, if a user asked the robot to place screw (i.e., the sequence node at depth 7 of Figure 6), then we cannot simply insert the sequence node, because its children need object pose as input, which is provided by the detect caddy node at depth 7. We thus need a way to not only directly insert the behavior node being asked, but find a self-contained node whose descendants provide the corresponding output ports.

ALGORITHM 8: Find Self-contained Behavior Node

Input: Node n (The node matching what is requested and $n.type \in \{Sequence, Subtree\}$)
Output: Node n_s (Self-contained execution or control node where keyed input parameters of its
  descendants are outputed from its descendants)

1   *// get all input ports possibly with duplicates*;
2   input_ports ← ∅;
3   **foreach** $e \in n.execution\_descendants$ **do**
4 | *input_ports ← input_ports ∪ e.input_ports*;
5   **end**

6   *// filter all dynamic, keyed input ports without duplicates*;
7   dynamic_input_ports ← select distinct * from input_ports where _.is_keyed();

8   *// find an ancestor node who or whose descendants output(s) all dynamic input ports*;
9   $n_s \leftarrow n$;
10   **foreach** $p \in dynamic\_input\_ports$ **do**
11 | **if** $n_s.execution\_descendants.has\_output\_port(p.type, p.key)$ **then**
12 | | continue;
13 | **else** *// go up a level*
14 | | $n_s \leftarrow n_s.parent$;
15 | | **while** $not(n_s.execution\_descendants.has\_output\_port(p.type, p.key))$ **do**
16 | | | $n_s \leftarrow n_s.parent$;
17 | | **end**
18 | **end**
19   **end**

20   **return** n_s

ALGORITHM 9: Append Self-contained Behavior Node As a Subgoal

---

Input: Node n (Current node in execution)
Input: Node n_s (Self-contained node, output from Algorithm 8)
Input: Node *root*

1   *// find the Subtree parent of "n" as current subgoal*;
2   $p \leftarrow n.parent$;
3   **while** $p \neq null$ **and** $p.type \neq Subtree$ **do**
4 | $p \leftarrow p.parent$;
5   **end**
6   assert($p \neq null$, *"must use Subtree as subgoal"*);
7   assert($p \neq root$, *"must have a Subtree as subgoal"*);

8   *// insert "n_s" after the current subgoal*;
9   root.insert_child_after(n_s, p);

Algorithm 8 shows the steps to find a self-contained behavior node n_s. First, we find the shallowest parent n of the action as the input of the algorithm. From lines 2–7, we get a unique set of the dynamic input ports from each execution descendant of n. From lines 9–19, we try to find all the output ports that provide data to the dynamic ports from n and, in the else block from lines 14–17, if n does not provide any dynamic input port, then we traverse the ancestors of n.

Algorithm 9 shows the steps to insert the self-contained node n_s after the current subgoal. Lines 2–5 find the current subgoal given the current node in execution n. Line 9 inserts n_s as a subgoal.

Before we move on to the next section, we have an important implementation note for practitioners. As the BTs are static, existing implementations may get the number of children before executing any child to check if every child is ticked. However, because we add the ability to dynamically insert a child, the number of children can be changed while a child is executing. During our implementation of Algorithms 8 and 9, we used a function to dynamically find the next sibling by locating the current running child and checking out of bound every time.

7 CASE STUDIES

In this section, we explore additional use cases of the algorithms presented in this article, including a gear insertion task for machining and explanations in the Taxi domain [15]. The two domains cover many applications in robotics, from manipulation to navigation, in continuous as well as discrete space.

7.1 Large Gear Insertion: A Machining Task

To evaluate the algorithms, we first apply them to the large gear insertion task, one that would be found in a manufacturing application. Note that we have also demonstrated the proposed algorithms on the screw picking and screw placing tasks, which we described earlier as examples to give readers a concrete idea next to the abstract discussions. As mentioned earlier, unlike the screw picking and placing tasks, the insertion task is a different task, as it is not a pick-and-place task but involves peg-in-hole insertion.

The behavior tree representation for the large gear insertion task is shown in Figure 9 and the framed tree is shown in Figure 10.

7.1.1 Hierarchical Explanation Generation. We first briefly review the algorithms to answer Q1–Q5. Answering Q1 is straightforward, which only needs to return the short description of the node in execution. Answering Q2 requires recursively visiting the ancestors of the current node n in execution, ignoring those without a name and those with the same name as n's, as well as decorator nodes. Answering Q3 needs to find the subtree parent of n, which may not always exist. Answering Q4 simply returns with the name of the root node. Answering Q5 involves performing a depth-first search for the descendants of n until it finds a child that has a name that is not the same as n's. During the process, it ignores all decorators and, when one of the children is found, the branch is not explored any further.

We now validate the generation algorithms by examining representative nodes in the large gear insertion task shown in Figure 9.

The simplest case is the `move to` node at depth 5 across all 3 subtasks. It has a `RetryUntilSuccesful` decorator parent and then a subgoal parent. In the context of the gear insertion subtask, the answers to Q1 and Q2 are "I move to see chuck" and "I move to see chuck in order to go insert large gear." In answering Q2, the `RetryUntilSuccesful` decorator is ignored. The answers to Q3 and Q4 are "My goal is to build a gearbox kit" and "My subgoal is to go insert large gear." When asked how the robot achieves the subgoal or goal, i.e., Q5, the answers are the following:

> To achieve the subgoal "go insert large gear," I need to do 3 steps. 1. move to see chuck. 2. detect chuck. 3. insert large gear.

> To achieve the goal "build a gearbox kit," I need to do 3 steps. 1. go insert large gear. 2. go pick screw. 3. go place screw.
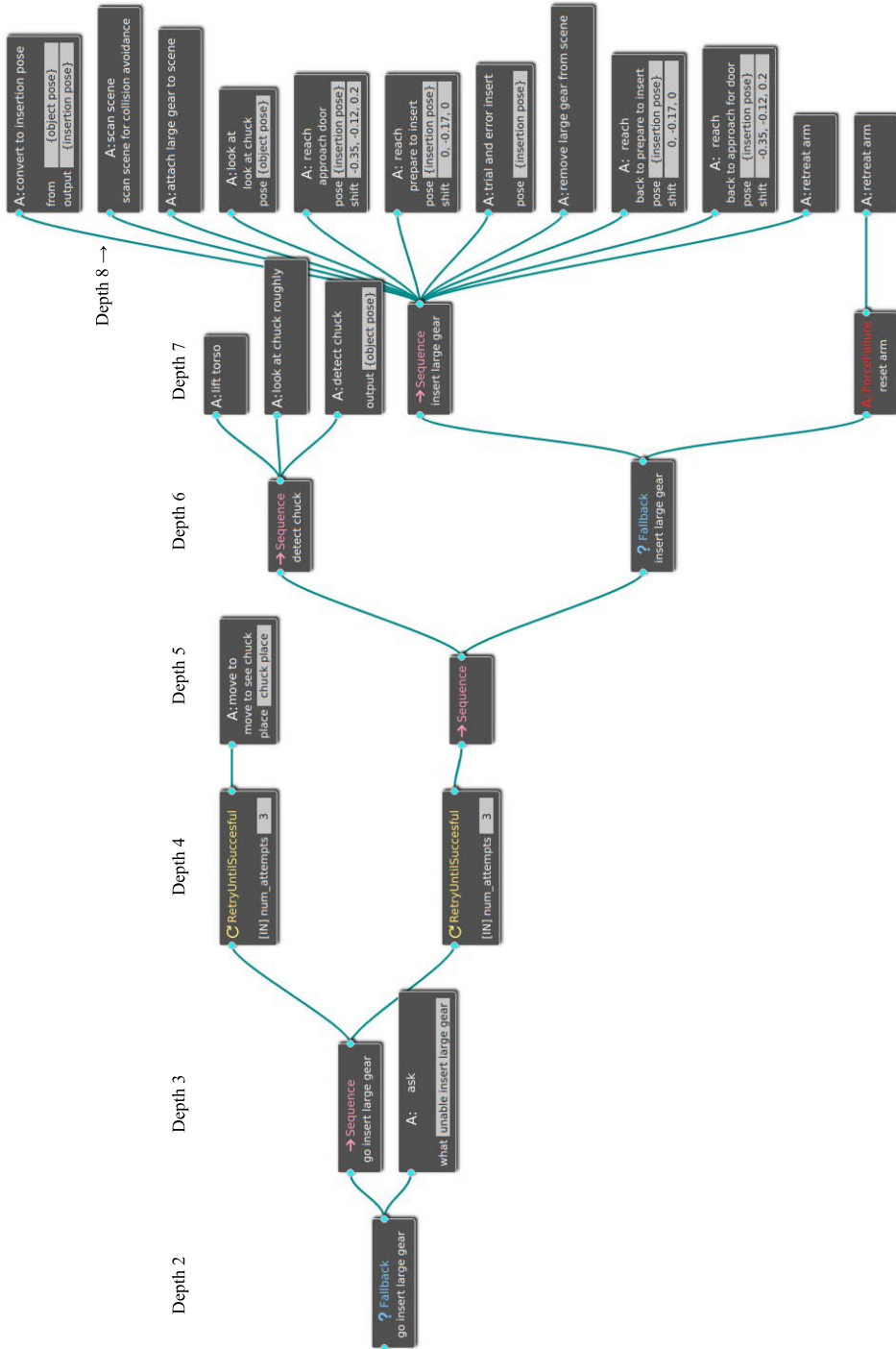
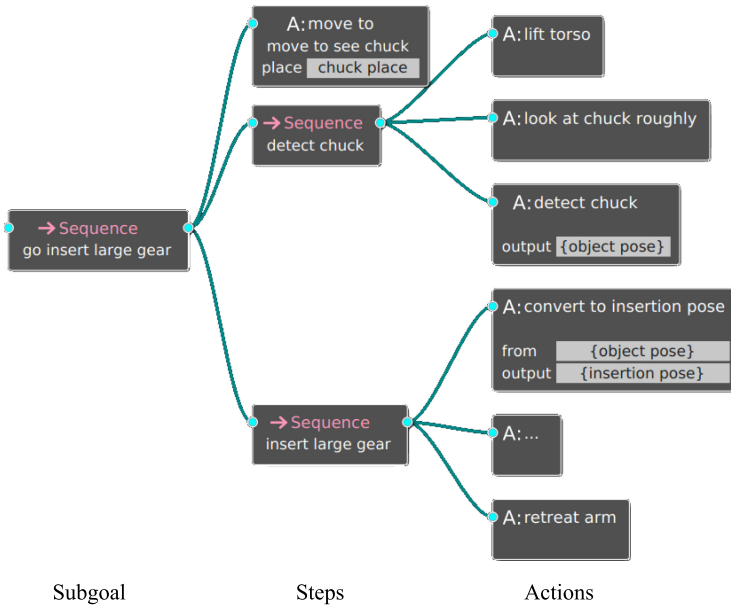Fig. 9. The large gear insertion subtask modeled in Behavior Trees.

Fig. 10. The framed BT for the large gear insertion subtask.

Note that the subgoal answer to Q5 is to verbally state the semantic set of "steps" shown in Figure 10.

A relatively deeper example in terms of tree depth are the `look at...` nodes at depth 7 across all 3 subtasks. It has an immediate sequence node parent that has a sequence parent without a name, because the `RetryUntilSuccesful` decorator parent can only have one child. In the context of the gear insertion subtask, the answers to Q1 and Q2 are "I look at chuck roughly" and "I look at chuck roughly in order to detect chuck." The answers to Q3–Q5 are the same, because the goal and the subgoal are shared.

Another example is from one of the execution nodes at depth 8. Let's take the `scan scene` node as the example. Still in the context of the gear insertion subtask, the answers to Q1 and Q2 are "I scan scene for collision avoidance" and "I scan scene for collision avoidance in order to insert large gear." The answer to Q3 is "My subgoal is to go insert large gear." The answers to Q4 and Q5 are the same as before.

With those three examples, one can easily infer the answers in the screw picking and place subtasks.

7.1.2 Dynamic Behavior Insertion as Subgoal. As we went through the insertion algorithm in the previous section, we will now consider how to answer the question "Can you insert large gear?" The `insert large gear` sequence node *i* at depth 7 will be first located. Then each of the descendants of *i*, from `convert to insertion pose` to `retreat arm` at depth 8 in Figure 9, will be examined to get a unique set of dynamic input ports: *{object pose, insertion pose}*. The *object pose* will be found from the `detect chuck` node (depth 7), a grandson of *i*'s grandparent, the empty sequence node (depth 5). The *object pose* input port will be found right at `convert to insertion pose`, which is a child of *i*. Thus, the empty sequence at depth 5 will be inserted after the subgoal node at depth 1 (Figure 4). The final tree is shown in Figure 11 with some nodes collapsed for readability.

The same logic also applies if a user asks "Can you place screw into caddy?" or "Can you pick screw?"
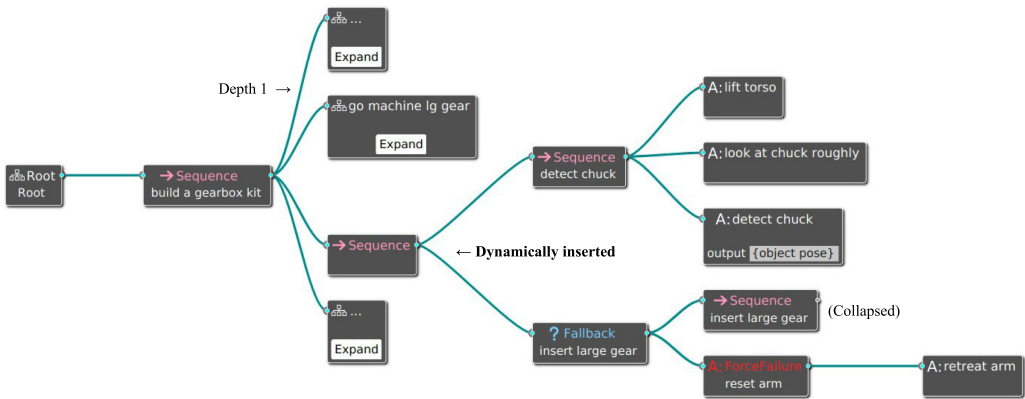
Fig. 11. The behavior tree for the kitting task after answering "Can you insert large gear?" Most subgoals in Figure 4 are collapsed for readability. The empty sequence node at depth 1 is dynamically inserted to ensure all input ports are satisfied by output ports. See Section 7.1.2 for more.

7.1.3 Explanations of Failures. Failures can happen at any node, so it is important to consider timing. Here, we will test Algorithm 7 in different scenarios. Again, we will use the behavior tree in Figure 9.

If the failure happened at the "look at chuck" node (fourth node at depth 8) and the question is asked while the robot is retreating its arm, then the answer will be "I could not insert large gear because look at chuck failed." The "insert large gear" is the description of the Fallback node (second node at depth 6). If the question is asked while "look at chuck" failed for the second time, then it will answer "I am retrying for attempt 1 to go insert large gear. I could not insert large gear because look at chuck failed." Here the retry information is added.

A failure can also happen during navigation, i.e., the "move to" node at depth 5. If the robot keeps navigating for a long time and a person wonders why, then the answer will be "I am retrying for attempt 1 to go insert large gear. I could not go insert large gear because move to see chuck failed."

Additionally, when the robot failed to detect chuck because it could not lift torso (first node at depth 7), the robot will answer "I am retrying for attempt 1 to go insert large gear. I could not detect chuck because lift torso failed."

If asked in the screw picking and place subtasks, the answers would be very similar.

7.2 Taxi Domain: A Navigation Task

While this work was originally designed for the context of the kitting task, we show here the potential of this work in a non-manipulation domain. In this modified Taxi domain [15], a taxi agent is tasked with picking up and dropping off a passenger while accruing into the fewest tolls possible.

We formalize this task as a deterministic Markov Decision Process, $MDP \coloneqq (S, A, T, R, \gamma, S_0)$, where S and A are state and action sets, respectively, $T : S \times A \rightarrow S$ is the transition function, $R : S \times A \times S \rightarrow \mathbb{R}$ is the the reward function, $\gamma \in [0, 1]$ is the discount factor, and S_0 is the initial state. Tolls provide negative reward, and delivering a passenger at the goal provides a large positive reward. The agent has an optimal policy $\pi^* : S \rightarrow A$ that maps states to actions that optimize the reward given an infinite horizon. An instance is shown in Figure 12.

The Taxi domain is implemented in PyGame[6] and the simple_rl framework [1]. We trained the agent using value iteration [48].
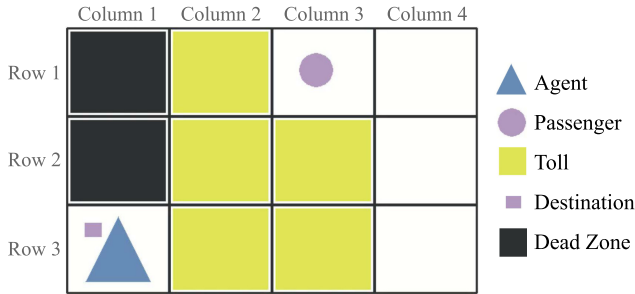
---

[6]https://www.pygame.org/.

Fig. 12. An environment of the Taxi domain, in which the agent delivers a passenger to the destination. Row and column numbers are added for easy reference.

The optimal policy π^* for the instance is represented in the behavior tree in Figure 13. With domain-specific goal and subgoals, the tree is semi-automatically generated by implementing listeners to state-action switches. For example, switching from moving in one of four directions to picking up the passenger will result in generating the `navigate to passenger` sequence.

The goal here is to deliver the passenger, which consists of two subgoals: go pick up passenger and go drop off passenger. To pick up the passenger, the agent first navigates to the passenger by moving all the way right (3 move `right` actions) and all the way up (3 move `up` actions) and takes a move `left` action before finally picking up the passenger. To drop off the passenger, the optimal agent takes the reverse path back to the destination. By taking the rightmost path, the agent avoids the toll in row 2, column 3.

As noted in the caption of Figure 13, the behavior tree representation is already in its simplified form, with no need for framing, such as removing decorator nodes. However, as shown below, the algorithms still apply without the use of decorator nodes as well as input and output ports. This shows that the behavior tree does not have to be complex to leverage the algorithms for explanation generation.

Similar to the large gear insertion task, we first examine the first action node move `right` at depth 4 in Figure 13. The algorithms can answer Q1 to Q5:

> Q1. What are you doing?
> I move right.
>
> Q2. Why are you doing this?
> I move right in order to navigate to passenger.
>
> Q3. What is your subgoal?
> My subgoal is to go pick up passenger.
>
> Q4. What is your goal?
> My goal is to go deliver passenger.
>
> Q5 (subgoal). How do you achieve your subgoal?
> To achieve the subgoal "go pick up passenger," I need to do 2 steps. 1. navigate to passenger. 2. pick up passenger.
>
> Q5 (goal). How do you achieve your goal?
> To achieve the goal "go deliver passenger," I need to do 2 steps. 1. go pick up passenger. 2. go drop off passenger.
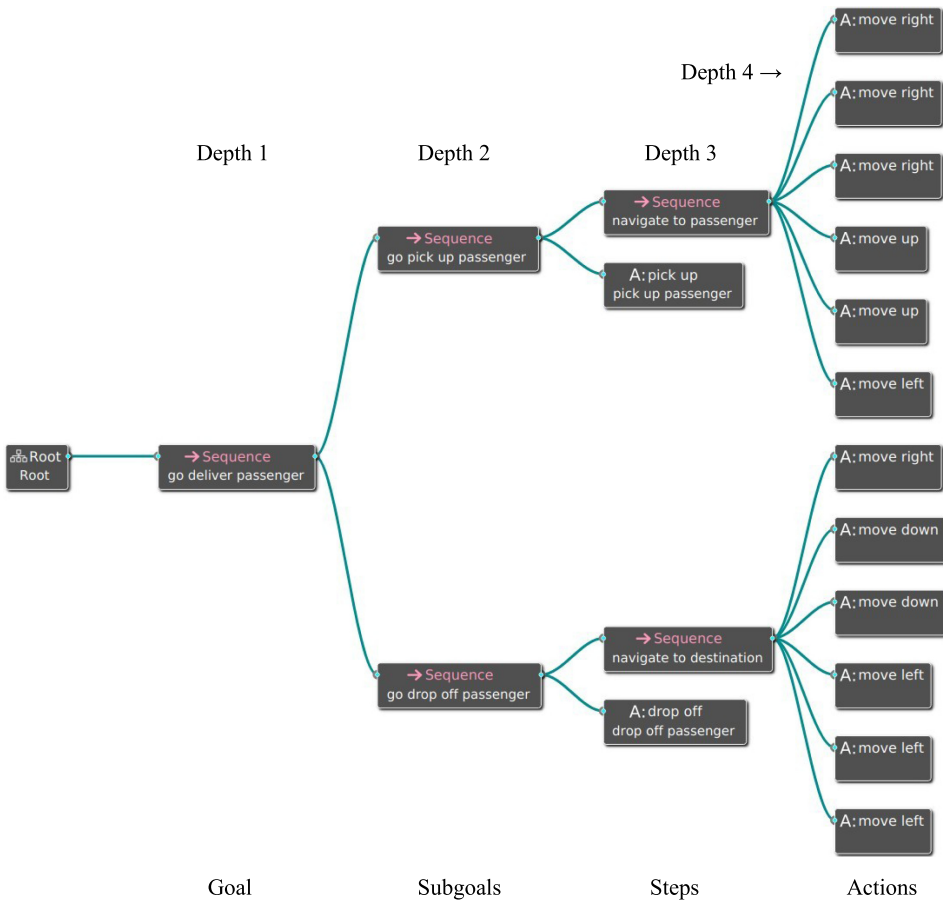
Fig. 13. The behavior tree representation of an optimal policy in the non-manipulation Taxi domain. Given that it is less complex than a kitting subtask, there is no need for simplification.

A different and more interesting example is the execution of the pick up or drop off action node at depth 3, which is not the deepest. Here, we will take the example of the drop off action node—the case for the pick up node is very similar. The following are the answers from the algorithms:

Q1. What are you doing?
I drop off passenger.

Q2. Why are you doing this?
I drop off passenger in order to go drop off passenger.

Q3. What is your subgoal?
My subgoal is to go drop off passenger.

Q4. What is your goal?
My goal is to go deliver passenger.

Q5 (subgoal). How do you achieve your subgoal?

To achieve the subgoal "go drop off passenger," I need to do 2 steps. 1. navigate to destination. 2. drop off passenger.

Q5 (goal). How do you achieve your goal?
To achieve the goal "go deliver passenger," I need to do 2 steps. 1. go pick up passenger. 2. go drop off passenger.

As the behavior tree does not have any input or output port dependencies, the dynamic behavior insertion as subgoal for the Taxi domain is trivial. We can insert the two subgoals with the following questions:

Can you go drop off passenger?
Can you go pick up passenger?

Using Algorithms 8 and 9, the corresponding subgoals represented by subtrees are inserted after the current subgoal.

7.2.1 Explaining Divergences between Behaviors. When a robot is executing its behavior tree, people's interpretations might be different, since humans tend to impute their own beliefs onto others [37]. After just a few executions, people will have often already formed a mental model of the robot's behavior. When this mental model differs from the robot's internal model, it is important to clarify where the divergences are and explain this discrepancy to avoid confusion.

An example can be drawn from the Taxi domain. Recall that Figure 13 shows an optimal policy where the toll in row 2, column 3 in Figure 12 was avoided to maximize the reward. After a few passenger deliveries in different environments, a person may still think that taking a *single* extra toll at row 2, column 3 is more reasonable than detouring through column 4 taking *three* extra steps to avoid the toll. They may not understand that 1 toll is more costly than 3 steps. In cases like this, it is beneficial to show where the divergences are to improve understanding.

Behavior trees can reveal such divergences automatically by presenting the robot's actual behavior and the human's expected behavior for easy comparison. As shown in Figure 14, the behavior tree representations of the two are placed side-by-side and the differences are highlighted in blue. The divergences were found using the sequence comparison algorithm proposed by Wu et al. [51]. The action sequences of the same subgoal between the two policies are compared by calculating the edit distance and **Longest Common Subsequence (LCS)**. Note that this visual comparison can also be used to differentiate behavior between multiple robots to improve comparative understanding, and we are currently implementing it to complement to the visualization in Figure 14.

8 LIMITATIONS AND FUTURE WORK

To the best of our knowledge, we are the first to explore Behavior Trees for robot explanation. We understand that there are still several limitations to the proposed work. We believe BTs are promising, but still require more work to satisfy the needs of a robust robot explanation system.

BTs have the ability to allow custom decorator nodes but framing cannot natively handle them, which might need to be explained when answering users' questions. A workaround solution is to have special cases in the algorithms, as in line 3 in Algorithm 2 and line 3 in Algorithm 5.

Also, to date, we have focused on the technical aspects, evaluating the systems with case studies for subtasks in the kitting task, a gear insertion task for machining, and within the Taxi domain; a user study is still needed with non-expert robot users to evaluate the level of understandability and the perception of the causal information in the answers to Q2–Q5. Having said this, our focus
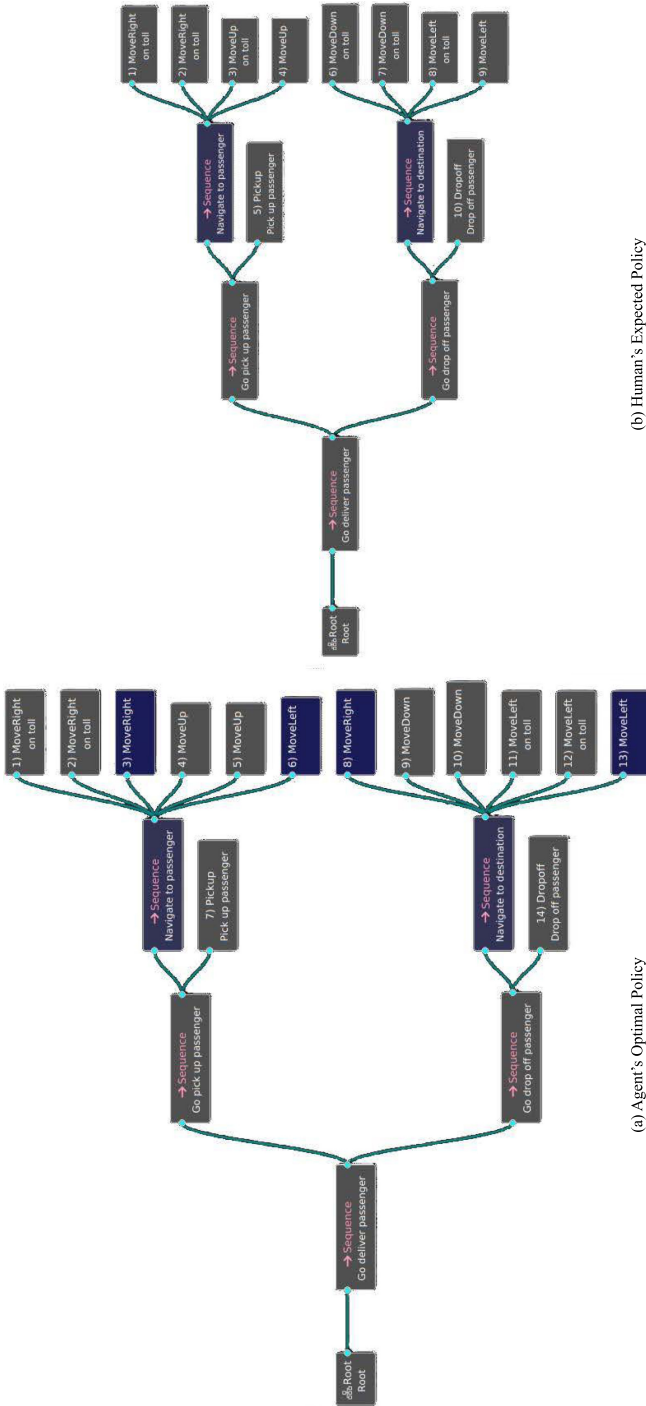
Fig. 14. An agent's optimal policy vs. a humans's expected policy for the environment in Figure 12. Divergences in actions (some leaf nodes) between the two policies are colored in dark blue and their immediate parents (some sequence nodes) are colored in blue–gray.

has been contributing robot generation algorithms in the scope of behavior trees, which paves the road to a robust explanation generation accounting for more human factors.

While we have demonstrated the use of our algorithms in manipulation domains and the Taxi domain, it remains relatively unknown how to support other non-manipulation tasks. For example, in a multi-robot system or a fleet of autonomous cars, robot explanations are increasingly needed given the increased complexity. However, given the complexity of robot manipulation and a sample use in a non-manipulation domain, we believe that the proposed algorithms can cover additional use cases, which will be explored in future work.

Another future goal is to investigate the questions being answered in this work; they are currently only analytically grounded but need to be validated in a user study. Currently, there is also a lack of research on what questions people would ask to get causal information on the behaviors that a robot is exhibiting. Like humans, are the "why" questions the only things humans seek for causal information from a robot's behavior? What about the "how" questions, such as "how do you achieve the goal?" What specific questions do people ask a robot generally? What causal questions do people ask a robot? These remain open research questions.

A drawback of BTs is that they do not generate a smooth, continuous arm trajectory if each waypoint is encapsulated in a behavior node separately. The recent development of the **Moveit Task Constructor (MTC)** [18] looks promising on this front. This library is a wrapper around the commonly utilized features of MoveIt such as motion planning and obstacle avoidance, but facilitates being able to track the current progress of execution, as well as makes specifying complicated tasks, such as pouring, easier. In MTC, manipulation primitives are represented as stages. Each stage can be linked together with other stages to form complete manipulation behavior. Examples of stages can be commands such as motion planning and moving the end effector to a specific pose or direct commands to the gripper joints. The success of each group of stages can propagate upwards as a reason for the success or failure of the behavior. For example, failure to reach a designated goal because of a detected collision would result in the behavior failing, because there is an obstacle blocking the path. These fine details, if used in the replies to the explanation questions, can further improve the understanding of the robot and its behaviors.

As MTC opens up the possibility for explanations at the low-level motion planning, we also need to explain low-level path planning in navigation, e.g., the popular ROS navigation stack [31]. Similar to MTC, we can enable the robot to explain if there is an obstacle on the ground blocking its path. Fortunately, we recently learned that the navigation stack in ROS 2 uses BTs to represent navigation behaviors [29], which paves the way for our explanation algorithms to be integrated into the navigation stack.

Last but not least, because BTs are also sequential, BTs do not support some properties of behavior execution that humans and animals are born with, such as pausing and reversing actions, which are the capabilities that a robot needs to have to achieve more natural, human-like robot explanation in certain circumstances.

9 CONCLUSION

In this article, we have demonstrated the use of Behavior Trees (BTs) for high-level robot explanations. We proposed framing BTs into a set of semantic sets, i.e., {the goal, subgoals, steps, actions}, and applied them to screw picking, screw placing, and large gear insertion manipulation tasks as well as the Taxi domain. We contributed algorithms to generate robot explanations, specifically giving answers to questions seeking causal information, which is what humans commonly seek from human explanations. We also described an algorithm that finds self-contained behavior based on the matching node from what is being asked, then inserts it after the current subgoal that the robot is achieving. We hope our work inspires other researchers working towards

the goal of transparent and trustworthy robots, and paves the road to a robust robot explanation system.

REFERENCES

[1] David Abel. 2019. simple_rl: Reproducible reinforcement learning in Python. In *Proceedings of the ICLR Workshop on Reproducibility in Machine Learning*. Retrieved from https://github.com/david-abel/simple_rl.

[2] Henny Admoni, Thomas Weng, Bradley Hayes, and Brian Scassellati. 2016. Robot nonverbal behavior improves task performance in difficult collaborations. In *Proceedings of the 11th ACM/IEEE International Conference on Human Robot Interaction (HRI'16)*. 51–58.

[3] J. Andrew Bagnell, Felipe Cavalcanti, Lei Cui, Thomas Galluzzo, Martial Hebert, Moslem Kazemi, Matthew Klingensmith, Jacqueline Libby, Tian Yu Liu, Nancy Pollard et al. 2012. An integrated system for autonomous robotics manipulation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'12)*. IEEE, 2955–2962.

[4] Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoğlu, and Georg Bartels. 2018. Know Rob 2.0 – A 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'18)*. IEEE, 512–519. Retrieved from https://github.com/knowrob/knowrob/.

[5] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. 2010. CRAM – A cognitive robot abstract machine for everyday manipulation in human environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'10)*. IEEE, 1012–1017.

[6] Gisela Böhm and Hans-Rüdiger Pfister. 2015. How people explain their own and others' behavior: A theory of lay causal explanations. *Front. Psychol.* 6 (2015), 139.

[7] Jonathan Bohren and Steve Cousins. 2010. The SMACH high-level executive. *IEEE Robot. Autom. Mag.* 17, 4 (2010), 18–20.

[8] Jonathan Bohren, Radu Bogdan Rusu, E. Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. 2011. Towards autonomous robotic butlers: Lessons learned with the PR2. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'11)*. IEEE, 5568–5575.

[9] Sebastian G. Brunner, Peter Lehner, Martin J. Schuster, Sebastian Riedel, Rico Belder, Daniel Leidner, Armin Wedler, Michael Beetz, and Freek Stulp. 2018. Design, execution, and postmortem analysis of prolonged autonomous robot operations. *IEEE Robot. Autom. Lett.* 3, 2 (2018), 1056–1063.

[10] Sebastian G. Brunner, Franz Steinmetz, Rico Belder, and Andreas Dömel. 2016. RAFCON: A graphical tool for engineering complex, robotic tasks. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'16)*. IEEE, 3283–3290.

[11] Michele Colledanchise and Petter Ögren. 2016. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Trans. Robot.* 33, 2 (2016), 372–389.

[12] Michele Colledanchise and Petter Ogren. 2018. *Behavior Trees in Robotics and Al: An Introduction (1st ed.)*. CRC Press, Boca Raton, FL.

[13] Munjal Desai, Poornima Kaniarasu, Mikhail Medvedev, Aaron Steinfeld, and Holly Yanco. 2013. Impact of robot failures and feedback on real-time trust. In *Proceedings of the 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI'13)*. 251–258.

[14] Sandra Devin and Rachid Alami. 2016. An implemented theory of mind to improve human-robot shared plans execution. In *Proceedings of the 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI'16)*. IEEE, 319–326.

[15] Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Intell. Res.* 13 (2000), 227–303.

[16] Anca D. Dragan, Kenton C. T. Lee, and Siddhartha S. Srinivasa. 2013. Legibility and predictability of robot motion. In *Proceedings of the 8th ACM/IEEE International Conference on Human-robot Interaction (HRI'13)*. 301–308.

[17] Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. 2019. Learning behavior trees from demonstration. In *Proceedings of the International Conference on Robotics and Automation (ICRA'19)*. IEEE, 7791–7797.

[18] Michael Görner, Robert Haschke, Helge Ritter, and Jianwei Zhang. 2019. MoveIt! task constructor for task-level motion planning. In *Proceedings of the International Conference on Robotics and Automation (ICRA'19)*. IEEE, 190–196.

[19] Kelleher R. Guerin, Colin Lea, Chris Paxton, and Gregory D. Hager. 2015. A framework for end-user instruction of a robot assistant for manufacturing. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'15)*. IEEE, 6167–6174.

[20] Pascal Haazebroek, Saskia van Dantzig, and Bernhard Hommel. 2011. A computational model of perception and action for cognitive robotics. *Cog. Proc.* 12, 4 (2011), 355.

[21] Zhao Han, Jordan Allspaw, Gregory LeMasurier, Jenna Parrillo, Daniel Giger, S. Reza Ahmadzadeh, and Holly A. Yanco. 2020. Towards mobile multi-task manipulation in a confined and integrated environment with irregular objects. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'20)*. IEEE, 11025–11031.

[22] Zhao Han, Jordan Allspaw, Adam Norton, and Holly A. Yanco. 2019. Towards a robot explanation system: A survey and our approach to state summarization, storage and querying, and human interface. In *Proceedings of the Artificial Intelligence for Human-Robot Interaction (AI-HRI) Symposium at AAAI Fall Symposium Series (AAAI-FSS)*. arxiv:1909.06418 (2019).

[23] Bradley Hayes and Julie A. Shah. 2017. Improving robot controller transparency through autonomous policy explanation. In *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction*. 303–312.

[24] Frank Kaptein, Joost Broekens, Koen Hindriks, and Mark Neerincx. 2017. Personalised self-explanation by robots: The role of goals versus beliefs in robot-action explanation for children and adults. In *Proceedings of the 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN'17)*. IEEE, 676–682.

[25] Minae Kwon, Sandy H. Huang, and Anca D. Dragan. 2018. Expressing robot incapability. In *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction (HRI'18)*. 87–95.

[26] Przemyslaw A. Lasota, Terrence Fong, Julie A. Shah, et al. 2017. A survey of methods for safe human-robot interaction. *Found. Trends Robot.* 5, 4 (2017), 261–349.

[27] Chong-U Lim, Robin Baumgarten, and Simon Colton. 2010. Evolving behaviour trees for the commercial game DEF-CON. In *Proceedings of the European Conference on the Applications of Evolutionary Computation*. Springer, 100–110.

[28] Tania Lombrozo. 2006. The structure and function of explanations. *Trends Cog. Sci.* 10, 10 (2006), 464–470.

[29] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. 2020. The Marathon 2: A navigation system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'20)*. IEEE, 2718–2725.

[30] Bertram F. Malle. 2006. *How the Mind Explains Behavior: Folk Explanations, Meaning, and Social Interaction*. The MIT Press.

[31] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. 2010. The office marathon: Robust navigation in an indoor office environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*. IEEE, 300–307.

[32] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. 2014. Towards a unified behavior trees framework for robot control. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'14)*. IEEE, 5420–5427.

[33] Grégoire Milliez, Raphaël Lallement, Michelangelo Fiore, and Rachid Alami. 2016. Using human knowledge awareness to adapt collaborative plan generation, explanation and monitoring. In *Proceedings of the 11th ACM/IEEE International Conference on Human Robot Interaction (HRI'16)*. IEEE, 43–50.

[34] AJung Moon, Daniel M. Troniak, Brian Gleeson, Matthew K. X. J. Pan, Minhua Zheng, Benjamin A. Blumer, Karon MacLean, and Elizabeth A. Croft. 2014. Meet me where I'm gazing: How shared attention gaze affects human-robot handover timing. In *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction (HRI'14)*. 334–341.

[35] Hirenkumar Nakawala, Paulo J. S. Goncalves, Paolo Fiorini, Giancarlo Ferringo, and Elena De Momi. 2018. Approaches for action sequence representation in robotics: A review. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'18)*. IEEE, 5666–5671.

[36] Hai Nguyen, Matei Ciocarlie, Kaijen Hsiao, and Charles C. Kemp. 2013. ROS Commander (ROSco): Behavior creation for home robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'13)*. IEEE, 467–474.

[37] Raymond S. Nickerson. 1999. How we know—and sometimes misjudge—what others know: Imputing one's own knowledge to others. *Psychol. Bull.* 125, 6 (1999), 737.

[38] Nils J. Nilsson. 1973. *A Hierarchical Robot Planning and Execution System*. SRI International.

[39] P. F. Palamara, V. A. Ziparo, L. Locchi, D. Nardi, P. Lima, and H. Costelha. 2008. A robotic soccer passing task using Petri net plans. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Demo Papers (AAMAS'08)*. International Foundation for Autonomous Agents and Multiagent Systems, 1711–1712.

[40] Chris Paxton, Andrew Hundt, Felix Jonathan, Kelleher Guerin, and Gregory D. Hager. 2017. CoSTAR: Instructing collaborative robots with behavior trees and vision. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'17)*. IEEE, 564–571.

[41] Chris Paxton, Felix Jonathan, Andrew Hundt, Bilge Mutlu, and Gregory D. Hager. 2018. Evaluating methods for end-user creation of robot task plans. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'18)*. IEEE, 6086–6092.

[42] Chris Paxton, Nathan Ratliff, Clemens Eppner, and Dieter Fox. 2019. Representing robot task plans as robust logical-dynamical systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'19)*.

[43] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: An open-source robot operating system. In *Proceedings of the ICRA Workshop on Open Source Software*. 5.

[44] Ismael Sagredo-Olivenza, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín, and Pedro Antonio González-Calero. 2017. Trained behavior trees: Programming by demonstration to support ai game designers. *IEEE Trans. Games* 11, 1 (2017), 5–14.

[45] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319.

[46] Jan Maarten Schraagen, Susan F. Chipman, and Valerie L. Shalin. 2000. *Cognitive Task Analysis*. Psychology Press.

[47] Martin J. Schuster, Sebastian G. Brunner, Kristin Bussmann, Stefan Büttner, Andreas Dömel, Matthias Hellerer, Hannah Lehner, Peter Lehner, Oliver Porges, Josef Reill, et al. 2019. Towards autonomous planetary exploration: The lightweight rover unit (LRU), its success in the SpaceBotCamp challenge, and beyond. *J. Intell. Robot. Syst.* 93, 3–4 (2019), 461–494.

[48] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. The MIT Press.

[49] Moritz Tenorth and Michael Beetz. 2009. KnowRob – Knowledge processing for autonomous personal robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'09)*. IEEE, 4261–4266. Retrieved from https://github.com/knowrob/knowrob/.

[50] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. 2016. Fetch & Freight: Standard platforms for service robot applications. In *Proceedings of the IJCAI Workshop on Autonomous Mobile Service Robots*.

[51] Sun Wu, Udi Manber, Gene Myers, and Webb Miller. 1990. An O (NP) sequence comparison algorithm. *Inf. Proc. Lett.* 35, 6 (1990), 317–323.

[52] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha. 2008. Petri net plans: A formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*. International Foundation for Autonomous Agents and Multiagent Systems, 79–86.