Building The Foundation of Robot Explanation Generation Using Behavior Trees

Zhao Han, Daniel Giger, Jordan Allspaw, Michael S. Lee (CMU), Henny Admoni (CMU), and Holly A. Yanco

Accepted to ACM Transactions on Human-Robot Interaction (THRI), 2021

March 31, 2021
Georgia Tech, Prof. Sonia Chernova
Motivation

Lack of explanations

- Unsettling & disturbing [Alison Gopnik, 2000]
- Explanations are natural in our daily life

Robots are increasingly capable yet complex

Improving understanding of robots

- Improves trust in real-time [Desai et al. HRI ’13]
- Leads to greater efficiency during more difficult human-robot collaboration scenarios [Admoni et al. HRI ’16]

Solved by human-agent/AI interaction community?

In a survey on explainable agents and robots, 47% used text-based communication methods [Anjomshoae et al. AAMAS ’19]

Big Picture & Focus

Step 1
Investigating Desired Robot Explanation

Step 2 (this talk)
Explanation Generation Using Behavior Tree

Step 3
Direct Explanation Communication: Projection Mapping Implementation

Step 4
Communicating Missing Causal Information by Replaying Past Actions

How to represent robot’s internal states and execution for explanation generation?
Background: Behavior Tree (BT)

An expressive action sequence method for hierarchical task specification and execution

Popular to model behaviors of AI agents in games

Gained momentum in robotics

- End-user programming -- Sawyer  
  (Colledanchise & Ogren, '18)

- Learning from demonstration & navigation in ROS 2  
  (French et al. ICRA '19)  
  (Macenski et al. ICRA '20)
Background: Behavior Tree (BT)

BT encodes behaviors through 2 node types

1. Control nodes

2. Execution nodes

<table>
<thead>
<tr>
<th>Notation</th>
<th>Node</th>
<th>Node Type</th>
<th>( s = \text{SUCCESS} )</th>
<th>( s = \text{FAILURE} )</th>
<th>( s = \text{RUNNING} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>Sequence</td>
<td>Control</td>
<td>All children ( \rightarrow ) ( s )</td>
<td>Any child ( \rightarrow ) ( s )</td>
<td>Any child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td>?</td>
<td>Fallback</td>
<td></td>
<td>Any child ( \rightarrow ) ( s )</td>
<td>All ( \rightarrow ) ( s )</td>
<td>Any child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td>⇒</td>
<td>Parallel</td>
<td></td>
<td>( M ) children ( \rightarrow ) ( s )</td>
<td>( N - M ) children ( \rightarrow ) ( s )</td>
<td>Any child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td>Diamond</td>
<td>Decorator</td>
<td></td>
<td>Function ( f ) ( \rightarrow ) ( s )</td>
<td>Function ( f ) ( \rightarrow ) ( s )</td>
<td>Function ( f ) ( \rightarrow ) ( s )</td>
</tr>
</tbody>
</table>

| Boxed   | Action   | Execution | Self \( \rightarrow \) \( s \) | Self \( \rightarrow \) \( s \) | Ticking |
| Circled | Condition|           | Self \( \rightarrow \) \( s \) | Self \( \rightarrow \) \( s \) | - |

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Color</th>
<th>Node</th>
<th>( s = \text{RUNNING} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>White</td>
<td>Subtree</td>
<td>Child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td></td>
<td>Pink</td>
<td>Sequence</td>
<td>Any child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>Fallback</td>
<td>Any child ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td></td>
<td>Varies</td>
<td>Decorator</td>
<td>Function ( f ) ( \rightarrow ) ( s )</td>
</tr>
<tr>
<td></td>
<td>White</td>
<td>Action (A)</td>
<td>Ticking</td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>Condition (C)</td>
<td>Ticking</td>
</tr>
</tbody>
</table>
## Why Behavior Tree

**Modularity, Reusability & Scalability**

- Loose coupling, easier to reuse subactions
- Allows the system to scale while maintaining fewer unique modules

**Simple** analogy of physical trees

- No extra abstraction
- Less loss while converting the underlying to explanations

---

## Other Methods

**State Machine**: intertwined transitions

- Flexible & well-known
- Challenging for complex tasks that are interconnected & involve multiple steps (complex dependencies)

**Peri Nets**: modeling concurrency and distributed execution

- Exposes technical details (e.g., operators, tokens)

(Colledanchise & Ogren, ’18)

(Nakawala et al., IROS ’18)
Modeling Robot Tasks Using Behavior Tree

Goal: assemble gearbox kits

- Navigate in a narrow workcell
- Collect complex parts

The Whole Kitting Task in Behavior Tree (Collapsed)
Screw Picking in BT

Node: [icon] Node ID
Node name
Input ports...
Output ports...

ID is required.
Others are optional.

Screw Placing in BT

A closer look:
- Nodes are modular and reused (½)
- Showed most node types
Cons of BT to Generate Explanations

As seen, tree structure is simple
- Minimal structural rules
- Yet as flexible as state machine

But final tree can be very deep
- Hard to justify
- Not suitable for human-preferred shallow explanations

When trees are shallower, it is easier for explanation algorithms

Behavior tree is also static
- Lacks ability of dynamic modification

After failure explanation, humans may ask
- “Can you pick another large gear?”
Framing BT for Hierarchical Explanation

Simplify and decompose the behavior tree of a multi-task, multi-step taskset into a set of semantic sets:

\{ the goal, subgoals, steps, actions \}.

Example: Screw picking subtask
Algorithms to Generate Hierarchical Explanation

Making BT interactive

- We can present BT for introspection
- With semantic sets, algorithms allow
  - user to ask
  - robot to explain hierarchically

Questions To Be Answered

Q1. **What** are you doing?
Q2. **Why** are you doing this?
Q3. What is your **subgoal**?
Q4. What is your **goal**?
Q5. **How** do you **achieve** your goal (or subgoal)?

Q2 gives the causal information of Q1.
Q3 gives the intermediate cause.
Q4 offers the final cause: “the end, function or goal”.
Q5 provides detailed steps to improve understanding.

Q2 to Q5: Causal knowledge people seek from explanations [1,2]


Similar to “context” in Das et al. HRI ’21

Explanation Generation Algorithm 1

Questions To Be Answered

Q1. What are you doing? Alg. 1
Q2. Why are you doing this? Alg. 2
Q3. What is your subgoal? Alg. 3
Q4. What is your goal? Alg. 4
Q5. How do you achieve your goal (or subgoal)? Alg. 5, 6

Algorithm 1: Answer “What are you doing?” (Q1)

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

1. // $p$.short_description $\leftarrow$ if $p$.has name then $p$.name else $p$.ID end;
2. return "I" + $p$.short_description + ",";

Algorithm 1 is trivial

It simply returns node name or ID
Explanation Generation Algorithm 2

Questions To Be Answered

Q1. What are you doing? Alg. 1
Q2. Why are you doing this? Alg. 2
Q3. What is your subgoal? Alg. 3
Q4. What is your goal? Alg. 4
Q5. How do you achieve your goal (or subgoal)? Alg. 5, 6

Algorithm 2: Answer “Why are you doing this?” (Q2)

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

1. $p \leftarrow n.parent$;
2. // find a non-decorator ancestor of “n” that has a name different from “n”;
3. while $p \neq \text{null or } p.type = \text{Decorator or } p.has.name() \text{ or } p.short.description = n.short.description do
4.   $p \leftarrow p.parent$;
5. end
6. return “I” + p.short.description + “ in order to ” + p.name + “.”;

Algorithm 2 reasons about current behavior

It finds the non-decorator control ancestor node with a different name
## Explanation Generation Algorithm 3

**Questions To Be Answered**

<table>
<thead>
<tr>
<th>Question</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1. <strong>What</strong> are you doing?</td>
<td>Alg. 1</td>
</tr>
<tr>
<td>Q2. <strong>Why</strong> are you doing this?</td>
<td>Alg. 2</td>
</tr>
<tr>
<td>Q3. <strong>What is your</strong> <strong>subgoal</strong>?</td>
<td>Alg. 3</td>
</tr>
<tr>
<td>Q4. <strong>What is your</strong> <strong>goal</strong>?</td>
<td>Alg. 4</td>
</tr>
<tr>
<td>Q5. <strong>How do you achieve</strong> your goal (or subgoal)?</td>
<td>Alg. 5, 6</td>
</tr>
</tbody>
</table>

### Algorithm 3: Answer “What is your subgoal?” (Q3)

**Input:** Node $n$ (Current node in execution)

**Output:** String `answer`

```
1 $p \leftarrow n$.parent;
2 \textbf{while} $p \neq \text{null and } p$.type $\neq \text{Subtree}$ \textbf{do}
3 \hspace{1em} $p \leftarrow p$.parent;
4 \textbf{end}
5 \textbf{if} $p \neq \text{null} \textbf{then return } "\text{My subgoal is to } + p$.name + ",";
6 \textbf{else return } "Sorry. I don’t have a subgoal.";
```

Algorithm 3 finds subtree ancestor as subgoal

Explanation Generation Algorithm 4

Questions To Be Answered

Q1. What are you doing? Alg. 1
Q2. Why are you doing this? Alg. 2
Q3. What is your subgoal? Alg. 3
Q4. What is your goal? Alg. 4
Q5. How do you achieve your goal (or subgoal)? Alg. 5, 6

Algorithm 4: Answer “What is your goal?” (Q4)

<table>
<thead>
<tr>
<th>Input: Node root</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: String answer</td>
</tr>
</tbody>
</table>

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

Algorithm 4 is also trivial

It simply uses the root node
Algorithm 5 & 6

Questions To Be Answered

Q1. What are you doing? Alg. 1
Q2. Why are you doing this? Alg. 2
Q3. What is your subgoal? Alg. 3
Q4. What is your goal? Alg. 4
Q5. How do you achieve your goal (or subgoal)? Alg. 5, 6

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 \( g = \text{null} \) 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

Depth-first search to find all steps: non-decorator nodes with unique names
Algorithms for **Failure** Explanation Generation

Framing makes explanations for the current behavior possible

Interesting to see explanations from failure handling

- “Was there anything wrong?”
- “What went wrong?”
- “How was the failure handled?”

**Idea:** Fallback & RetryUntilSuccessful nodes
Algorithms for Failure Explanation Generation

**Fallback node (Preferred):**
- Find a FAILURE child node

**RetryUntilSuccessful node**
- Check attempt number of a node
- Parent used to indicate what is being attempted

Then we can also combine both

---

```java
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 ← n.parent;
2. is.wrong, fell.back ← false;
3. while p ≠ null and p.type ≠ Subtree and is.wrong ≠ true do
   4. if p.type = Fallback and p.children[0].is.wrong ≠ true then // Fallback node
      5. is.wrong, fell.back ← true;
      6. answer ← “I could not” + p.short.description + “ because “;
      7. n_fal ← DepthFirstSearch(p, f(node)) → begin
      8. if node.failed and node.type ∈ {Condition, Action} then return node;
      9. end
      10. if n_fal.parent ≠ null and n_fal.parent.short.description = p.short.description then
           11. answer += “I was unable to” + n_fal.parent.short.description + “ as “;
           12. answer += n_fal.short.description + “ failed.”;
       13. else if p.type = Retry and p.attempt ≠ 0 then // Retry node
           14. is.wrong ← true;
           15. rp = p.find.non.null.parent;
           16. if rp ≠ null then
               17. answer = “I am retracting for attempt” + p.attempt + “ to” + rp.short.desc + “”;
               18. n_fal ← DepthFirstSearch(p, f(node)) → begin
               19. if node.failed and node.type ∈ {Condition, Action} then return node;
               20. end
               21. fp = n_fal.first.ancestor.with.name;
               22. answer += “I could not” + fp.short.desc + “ because” + n_fal.short.desc + “ failed.”;
               23. p ← p.parent
           24. end
       25. if fell.back then // Check if the Fallback node has Retry parent
           26. p ← fallback.node.parent;
           27. while p ≠ null and p.type ≠ Subtree do
           28. if p.type = Retry and p.attempt ≠ 0 then
               29. rp = p.find.non.null.parent;
               30. if rp ≠ null then
                   31. n = “I am retracting 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;
```
Dynamic Behavior Insertion as Subgoal

BT still has dependencies

When asked to place another screw

- Cannot simply insert the “place screw” sequence
- Need to go pick a screw first

Approach: Find self-contained node with descendants satisfying output ports
Dynamic Behavior Insertion as Subgoal

Algorithm to find self-contained node with children satisfying output ports

Input: Requested Behavior Node n

1. **Find all dynamic input ports** from each execution descendant of n

2. Traverse ancestors to **find all output ports** that provides data to the input ports

```
Algorithm 8: Find Self-Contained Behavior Node

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

1 // get all input ports possibly with duplicates;
2 input.ports ← Ø;
3 foreach e ∈ 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 ns ← n;
10 foreach p ∈ dynamic_input.ports do
11      if ns.execution_descendants.has_output_port(p.type, p.key) then
12         continue;
13      else // go up a level
14         ns ← ns.parent;
15         while not(ns.execution_descendants.has_output_port(p.type, p.key)) do
16            ns ← ns.parent;
17        end
18      end
19 end
20 return ns
```
Case Studies

Additional use cases

1. Large Gear Insertion: A Machining Task

- In addition to screw pick & place tasks
- Different: Involves peg-in-hole insertion
  - Manufacturing application
Case Studies

Additional use cases

1. Large Gear Insertion: A Machining Task
   - In addition to screw pick & place tasks
   - Different: Involves peg-in-hole insertion
     - Manufacturing application

What the algorithms see: subgoal, steps, actions
Large Gear Insertion: A Machining Task

Q1. What are you doing? I move to see chuck.

Q2. Why are you doing this? I move to see chuck in order to go insert large gear.

Q3. What is your subgoal? My subgoal is to go insert large gear.

Q4. What is your goal? My goal is to build a gearbox kit.

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

Q5 (goal). How do you achieve your goal? To achieve the goal “build a gearbox kit”, I need to do 3 steps. 1. go insert large gear. 2. go pick large gear. 3. go place large gear.

Answer for “look at chuck” failure I could not insert large gear because look at chuck failed.

Answer for second “look at chuck” failure I am retrying for attempt 1 to go insert large gear. I could not insert large gear because look at chuck failed.
Case Studies

2. Taxi Domain: A Navigation Task

(Dietterich, JAIR ‘20)

Discrete space

Represent reinforcement learning agent’s optimal policy in BT:

- Semi-automatically generated with listeners to state-action switches

No framing: shows our work applies to less complex tasks

Explaining Divergences Between Behaviors

Lastly, people form mental models of others, including robots.

Mental model can be different from a robot’s execution of the BT.

Important to avoid confusion:

- Clarify where divergences are
- Explain this discrepancy

Key Takeaways

Behavior Tree (BT) is modular, reusable, and scalable (compared to interconnected state machine).

For shallow, hierarchical explanation, we frame BT into \{goal, subgoals, steps, actions\}.

- This makes explanation generation algorithms simple, which is demonstrated in different types of FetchIt tasks and the taxi domain.

By finding self-contained nodes, our algorithm allows dynamic behavior insertion into BT.

For failure explanation generation, we leveraged Fallback and RetryUntilSuccessful nodes.

Last but not least, we also explored explaining the gap between human’s mental model & robot’s execution.
Thanks!

Open source  github.com/uml-robotics/robot-explanation-BTs

Zhao Han  zhao_han@student.uml.edu

UMass Lowell Human-Robot Interaction Lab: robotics.cs.uml.edu

Building The Foundation of Robot Explanation Generation Using Behavior Trees

Zhao Han, Daniel Giger, Jordan Allspaw, Michael S. Lee (CMU), Henny Admoni (CMU), and Holly A. Yanco