Fetch Me a Beer: A Behavior Tree Walkthrough for ROS 2 Developers

Audience: ROS 2 developers with moderate skill and basic Behavior Tree familiarity
BT Source: `can_do_challenge/bt_xml/main.xml`

This article assumes you have knowledge of the BehaviorTree.CPP library. This describes the architecture of the behavior tree I developed as a first step towards participating in the Robotics Society of Southern California Can-do Challenge. The tree is not ready for prime time; it is just a proof of concept. Another paper discusses the work that went into crafting this tree and all the pieces needed to test this in simulation.

Overview

The "fetch me a beer" task is a compact stress test for autonomy: the robot must navigate, perceive, manipulate, and remain safe in a dynamic environment. The behavior tree in `main.xml` solves this by **decomposing the mission into subtrees** and using **reactive control flow** to keep safety and perception live at all times.

What to copy if you're building your own

If you're attempting the Can-Do Challenge with a different robot, these are the parts worth reusing even if you ignore everything else:

- **Reactive safety subtree**: Keep safety checks live on every tick.
- **Blackboard flags**: Simple, explicit gates like `needToNavigate` and `needToGrasp` prevent repetitive behavior.
- **Two-camera handoff**: Long-range detection (OAK-D) to get close, short-range detection (Pi camera) for final alignment.
- **Visual servo loops**: Small, repeated corrections beat complex one-shot plans.

Assumptions baked into this tree

This design only works under a few assumptions. If these aren't true for your robot, the tree will struggle:

- The can is on a flat table of roughly known height.
- The expected can pose is "close enough" to be visible to the OAK-D from the initial navigation goal.
- The gripper can reach the can once the base is within ~0.6m.
- The Pi camera's frame is reliable for fine alignment at short distances.

Big picture: the mission tree

The `Root` captures the mission flow:

1. Save the start pose so the beer can be returned to me.
2. Set up various variables.
3. Fetch the can (explained below).
4. Back away from the table to ensure clearance before returning.
5. Return to the start pose and bring the beer back to me.

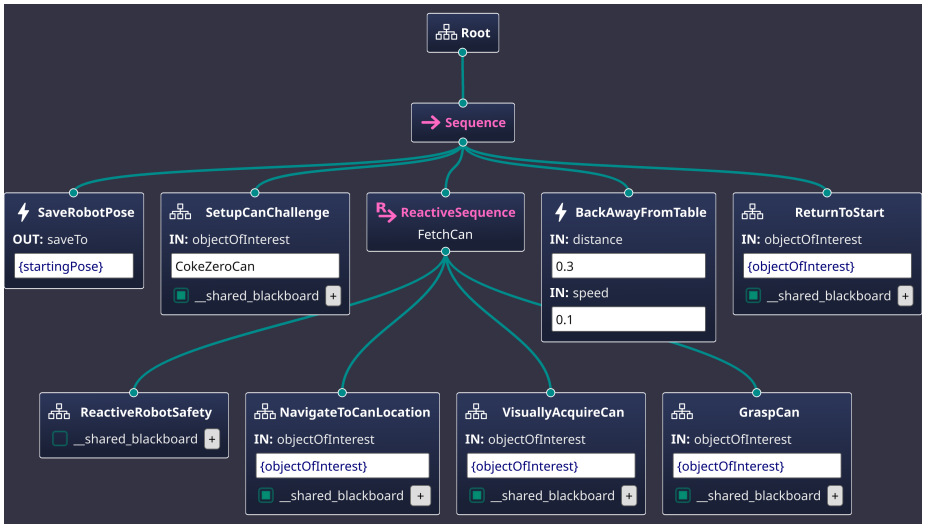

Figure 1: Tree

Key point: reactive execution

The `FetchCan` node is a **ReactiveSequence**. This means **every tick re-evaluates all children**. If perception changes or safety triggers, the tree responds immediately without being "stuck" in an old action.

Safety: always-on, always first-class

Safety is implemented as its own subtree (`ReactiveRobotSafety`) using a **ReactiveSequence** of checks:

- `RobotHasNotFallenOver`
- `RobotIsNotAboutToFallOver`
- `RobotIsNotEStopped`
- `EnsureBatteryIsNotCritical`
- `BatteryIsCharged`

Each of those uses **Inverter + ReactiveFallback** patterns to convert a bad condition into a shutdown or E-stop action. In practice, each test will be

transformed into a reactive fallback node so that, say, if the robot has fallen over the whole system will shut down after sending me an alert message, but if the robot is only about to fall over the system will signal an emergency stop (e-stop) to the robot and, again, send me a message.

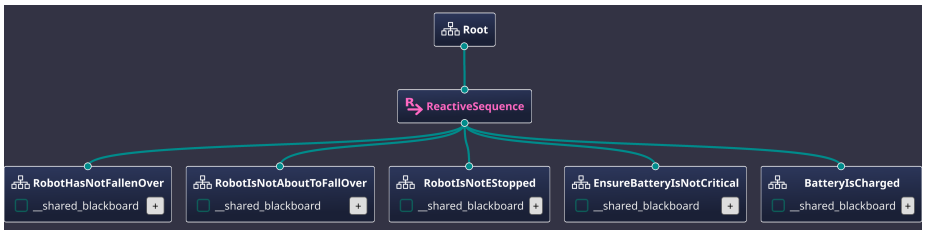

Figure 2: ReactiveRobotSafety

The safety system is just a stand-in for the moment, it's not really participating yet in the full safety system for the robot ***Sigyn*** as I described in another paper. One of the purposes of this tree was to test reactive safety.

Because `ReactiveRobotSafety` is a reactive sequence, and the whole tree is designed to be reactive, at every tick of the tree, safety will be tested and dealt with.

Getting to near where we think there might be beer

Looking back at the `NavigateToCanLocation` node and its expansion, we use a reactive fallback node with a guard test to make sure that we only try to get to the can once. The `SetupCanChallenge` node sets the blackboard `needToNavigate` variable to `true` and, once we have achieved the goal of getting to the can, we set `needToNavigate` to `false` so it won't be tried again. Once more, this is not a behavior tree that is ready for production, as that would deal with a situation, say, where we found a can of beer and then found the can was empty so we would need to go find a different can. But, for the challenge, this suffices.

The `ComputePathToCanLocation` and `NavigateToPoseAction` nodes are straight out of the Nav2 stack's behavior tree to navigate to some pose. We only need to add at the end a node to set `needToNavigate` to false as just described.

Having seen a can of beer, approach it

Looking back at the `VisuallyAcquireCan` node and its expansion, we again use a reactive fallback node with a guard test to make sure that we only approach the can once. The `SearchForCanWithOAKD` subtree runs code on a separate thread and does object detection of the can. The `ApproachCanWithOAKD` subtree moves the robot close enough that the gripper could, theoretically, reach the can. Once more, there is no useful recovery behavior here. For instance, the initial movement of the robot looks up in a database to see possible locations of

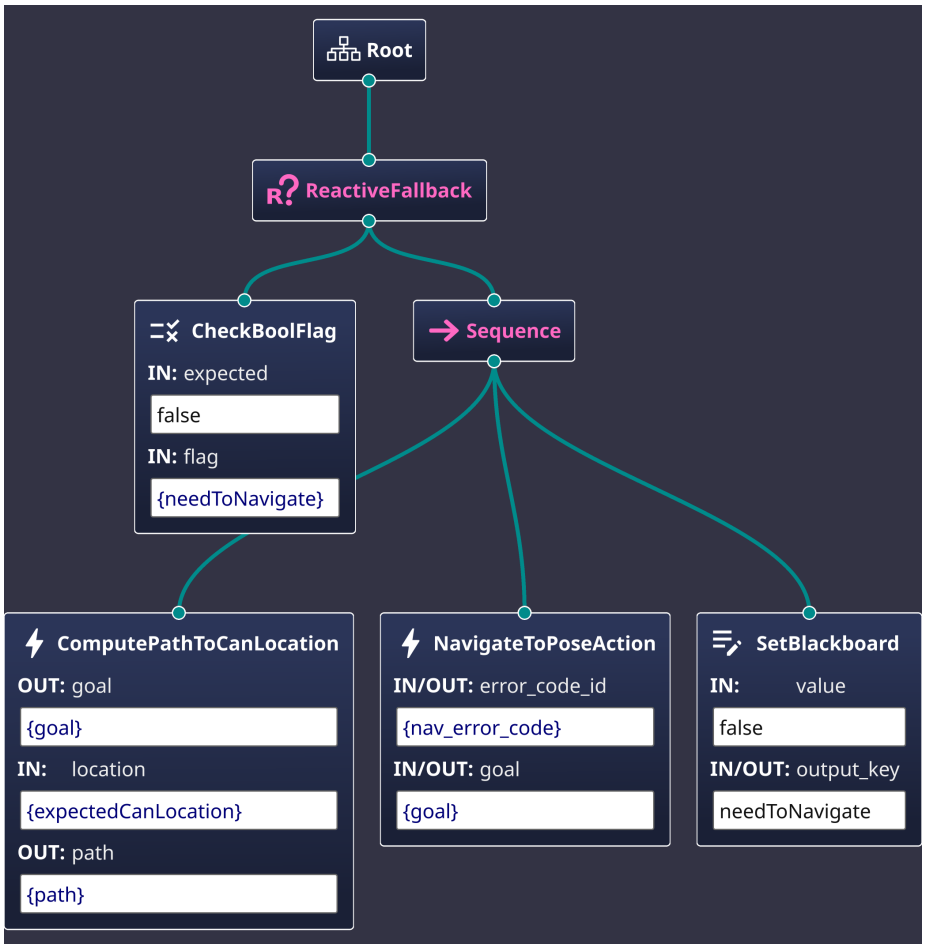

Figure 3: NavigateToCanLocation

beer—maybe there is some in the refrigerator, maybe there's one on the kitchen table. The database gives a pose where the expectation is that the OAK-D camera should see a can of beer if one is there.

In fact, the OAK-D has a fairly narrow field of view. If the can is even one inch outside of that field of view, no attempt is made to look around to see if the can is nearby. And even if the can is seen, there is a simple attempt to get near the can—there is no recovery if, for some reason, there is something in the way. A real behavior tree will have to deal with possible failures and provide recovery behavior.

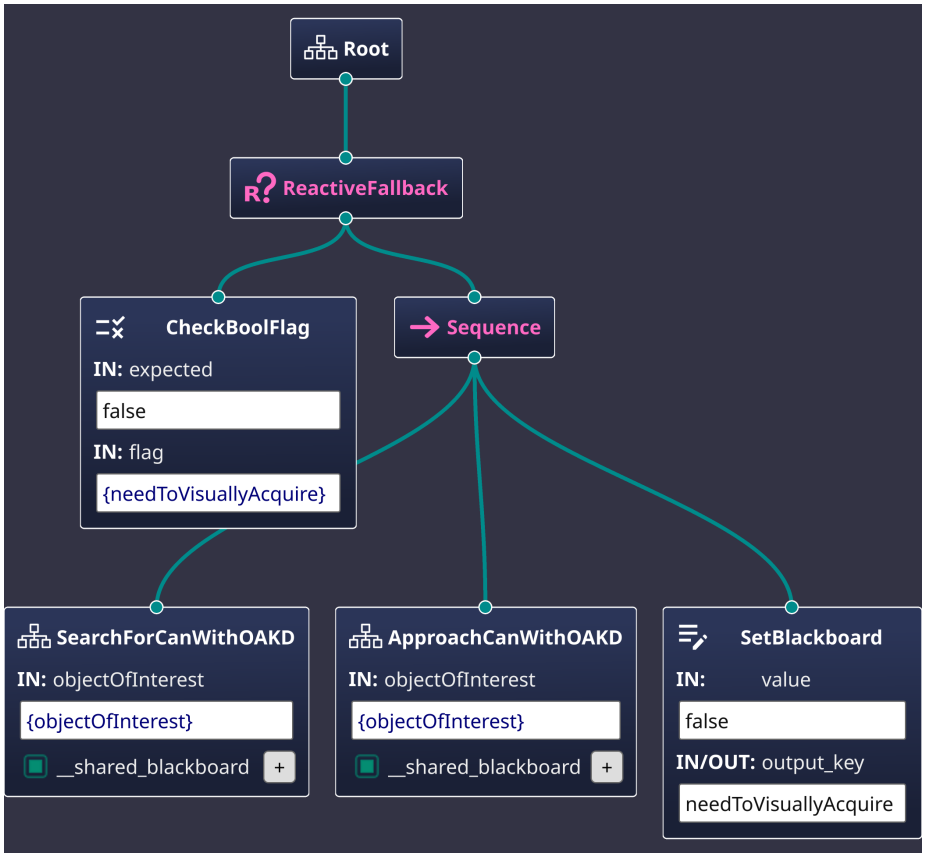

Figure 4: VisuallyAcquireCan

Let's keep going down the behavior tree and look at the subtree expansion of the `SearchForCanWithOAKD` node.

All that is implemented here is a typical backchain design pattern. The reactive fallback node has as its first child a condition node that wants to check for the desired state and the second child creates the desired state. There is a custom

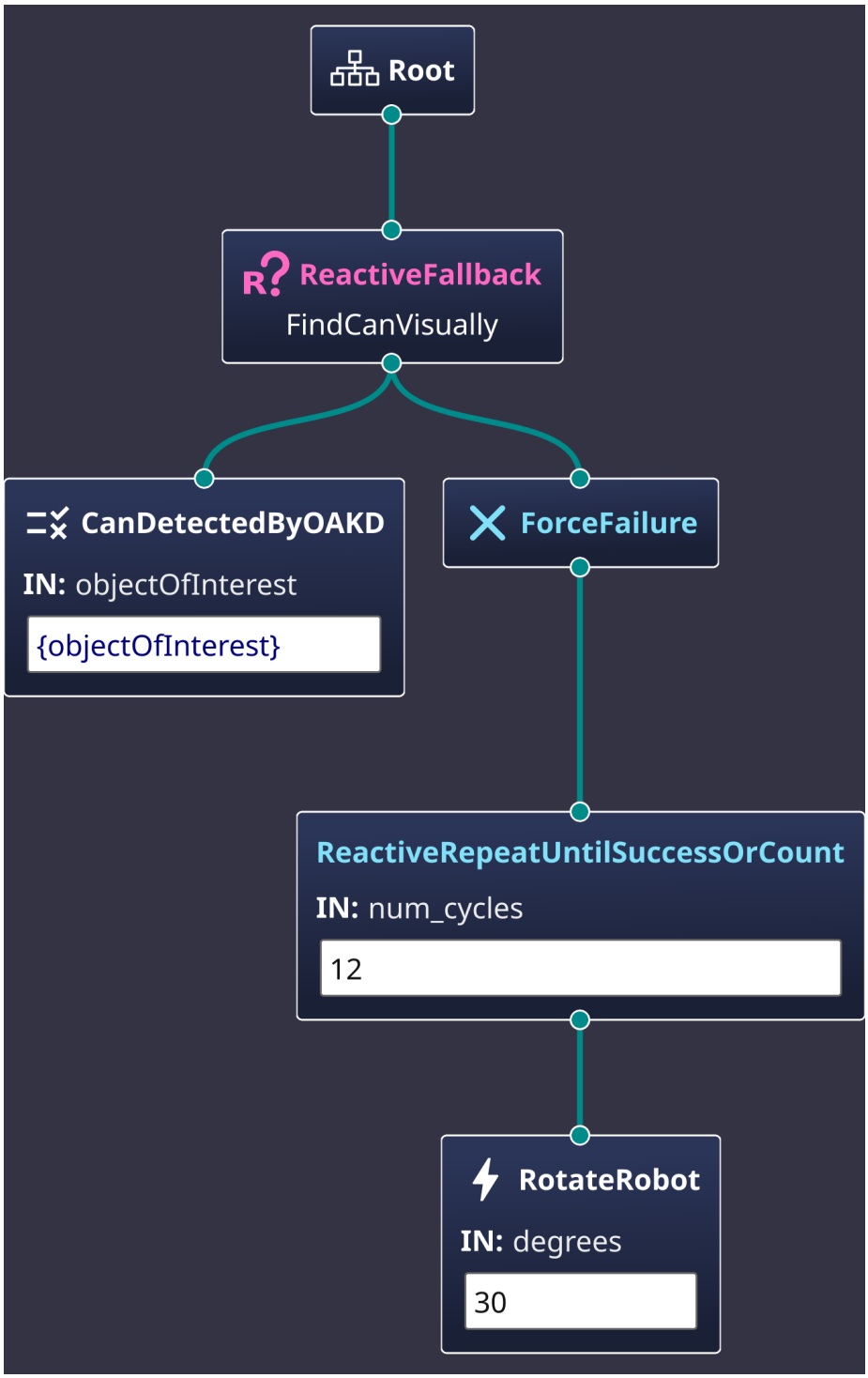

Figure 5: SearchForCanWithOAKD

node, `CanDetectedByOAKD`, that will return SUCCESS if the object detector for the OAK-D camera sees the can. If not, the second child node will rotate the robot 30 degrees, up to 12 times. This is not sufficient for a real behavior tree but, again, it suffices for the challenge. The assumption is that we have moved to where the can should be visible. The goal is to put the can roughly centered horizontally in the image at a place where, if we were to directly drive towards that place, we would be in a place that the Pi camera, with its limited field of view, especially given where it is mounted on the robot, should be able to also see the can (assuming the camera was raised up to the same level as the can).

Since we do not yet know that the OAK-D camera can see the can, the strategy here is admittedly primitive: we just rotate the robot until it does.

The `SearchForCanWithOAKD` subtree uses a simple rotate-and-check approach. If `CanDetectedByOAKD` returns FAILURE (meaning the object detector sees nothing), the `ForceFailure` branch kicks in. This triggers `RotateRobot`, which commands a 30-degree turn. Because this is wrapped in a `ReactiveRepeatUntilSuccessOrCount` decorator, the robot will incrementally spin up to 12 times (a full 360 degrees) until the detection condition flips to SUCCESS.

It's basic, but for a single-room challenge where we are roughly looking at the table, it works.

Closing the Distance

Once the OAK-D has a fix on the target, we switch to the `ApproachCanWithOAKD` subtree. This is where the rubber meets the road—or rather, where the wheels meet the floor.

We use a classic **Reactive Fallback** pattern here named `GetWithinReach`. The goal is to satisfy `CanWithinReach`, which checks if the beer is within 0.6 meters. If we aren't close enough, we execute the fallback branch.

Here we see a specialized custom action: `MoveTowardsCan`. This node is a bit of a hybrid. Unlike standard Nav2 nodes that plan a global path around obstacles, `MoveTowardsCan` acts more like a visual servo. It grabs the "fresh" detection from the OAK-D (ignoring anything older than 0.25 seconds), calculates the angle error, and steers the robot directly at the target.

If the robot is too far off-angle (> 0.1 radians), it rotates in place. If it's aligned, it drives forward proportional to the distance. If the can is too close for a safe rotate, it backs up first. It's a reactive controller that bypasses the global planner for the "last mile" approach.

Standard Nav2 navigation (`ComputeApproachGoalToCan` -> `NavigateToPoseAction`) is essentially a backup here. If `MoveTowardsCan` fails or isn't applicable, we fall back to standard path planning. The `ComputeApproachGoalToCan` node does some nice math: it takes the detection, transforms it into the `base_link` frame

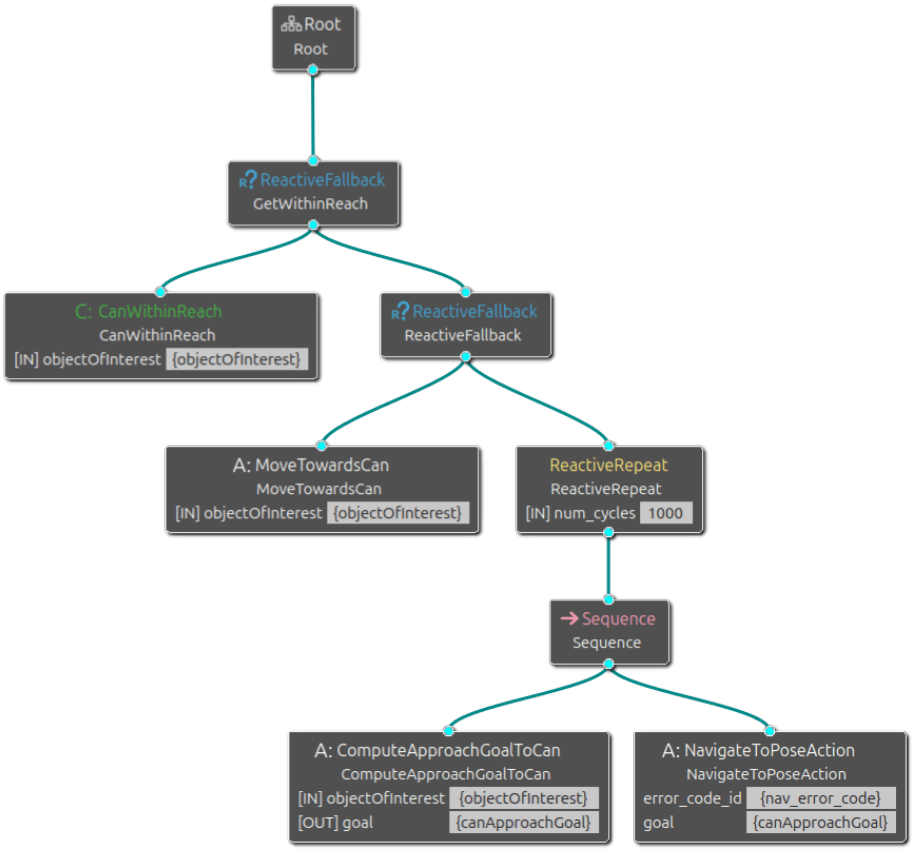

Figure 6: ApproachCanWithOAKD

to get a bearing, determines a goal pose 0.6m away from the can, and then transforms that goal into the `map` frame for Nav2 to digest.

More precisely, the computed goal is **base_link-relative** first. The code only translates forward when the bearing is within roughly 20°, and it caps the forward step to about 0.5m. This prevents the robot from surging forward on noisy detections and keeps the motion incremental.

Node behavior cheat sheet (for builders)

If you're reading the XML and wondering what the custom nodes really do, here are the short summaries:

- **`MoveTowardsCan`**: Visual servo with freshness gating; rotates if off-angle, backs up if too close to rotate, drives forward only on fresh detections.
- **`ComputeApproachGoalToCan`**: Builds a base_link-relative pose from the bearing, then transforms to map for Nav2; translates only when roughly facing the can.
- **`AdjustExtenderToCenterCan`**: Micro-rotations based on Pi camera x-offset, with minimum angular speed to overcome stiction.
- **`ElevatorAtHeight`**: Uses Pi camera to check Z alignment and iterates 2cm steps until within ~1.8cm.

The Complex Part: Grasping

Now for the main event. **`GraspCan`** is the most complex subtree because it has to coordinate the elevator, the arm extender, and the gripper, all while keeping the can centered in the camera view.

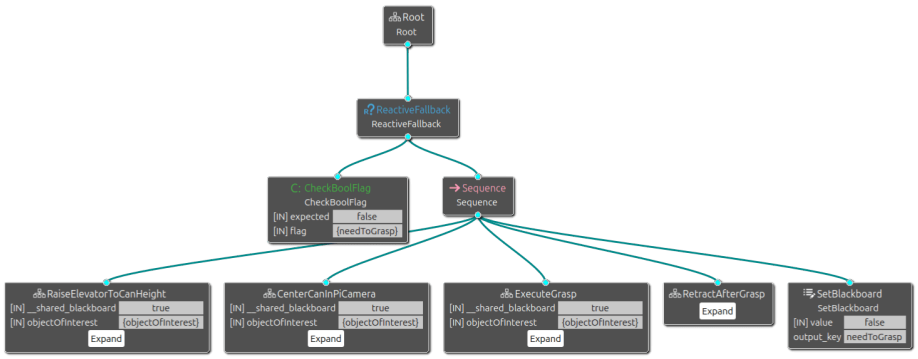

Figure 7: GraspCan

This subtree uses a `ReactiveFallback` to check a blackboard flag `needToGrasp`. If true, it enters a sequence of operations that looks like a simplified state machine.

1. Elevator Alignment

First, we have to get the gripper to the right height. This isn't as simple as commanding "Go to 0.7 meters."

The `RaiseElevatorToCanHeight` subtree uses a **polling loop** (`RetryUntilSuccessful`). Inside, it checks `ElevatorAtHeight`. This condition node takes the detection from the **Pi Camera** (which is mounted on the gripper assembly itself), transforms it to the `base_link` frame, and compares the can's Z-height against the gripper's Z-height.

If they don't match (within a tight 1.8cm tolerance), `StepElevatorUp` nudges the elevator up by 2cm. We force a failure to make the loop repeat. This allows the system to visually "climb" until it's level with the can, compensating for any uneven floor or odometry drift.

2. Fine Center Alignment

Once the height is right, we need horizontal precision. The OAK-D got us close, but the Pi Camera on the arm provides the final alignment.

`CenterCanInPiCamera` uses a retry loop. The `AdjustExtenderToCenterCan` action is interesting: it looks at the horizontal pixel offset (x-axis) in the Pi Camera. If the can is off-center, it calculates the exact rotation angle needed (`atan2(x_offset, distance)`) and pulses the robot's base rotation.

The code actually has logic to "boost" the rotation speed if the required movement is too subtle for the motors to overcome stiction, ensuring we don't just stall trying to make a 0.5-degree correction.

3. The Physical Grasp

With vertical and horizontal alignment locked in, we execute the grasp.

The `ExecuteGrasp` sequence is fairly linear: 1. **OpenGripper**: Wide open. 2. **ExtendTowardsCan**: The arm extends. The code calculates the extension distance as `can_distance - 0.045m`. That 4.5cm offset is crucial—it positions the gripper fingers *around* the can rather than smashing *into* it. 3. **CloseGripperAroundCan**: It calculates the servo positions for a specific can diameter (66mm). 4. **StepElevatorUp**: We lift the can 10cm straight up to clear the table. 5. **VerifyGrasp**: We check the camera again. If the can is gone, we dropped it.

4. Retract

Finally, we back out.

`RetractExtender` pulls the arm back to zero. `LowerElevatorSafely` brings the center of gravity down.

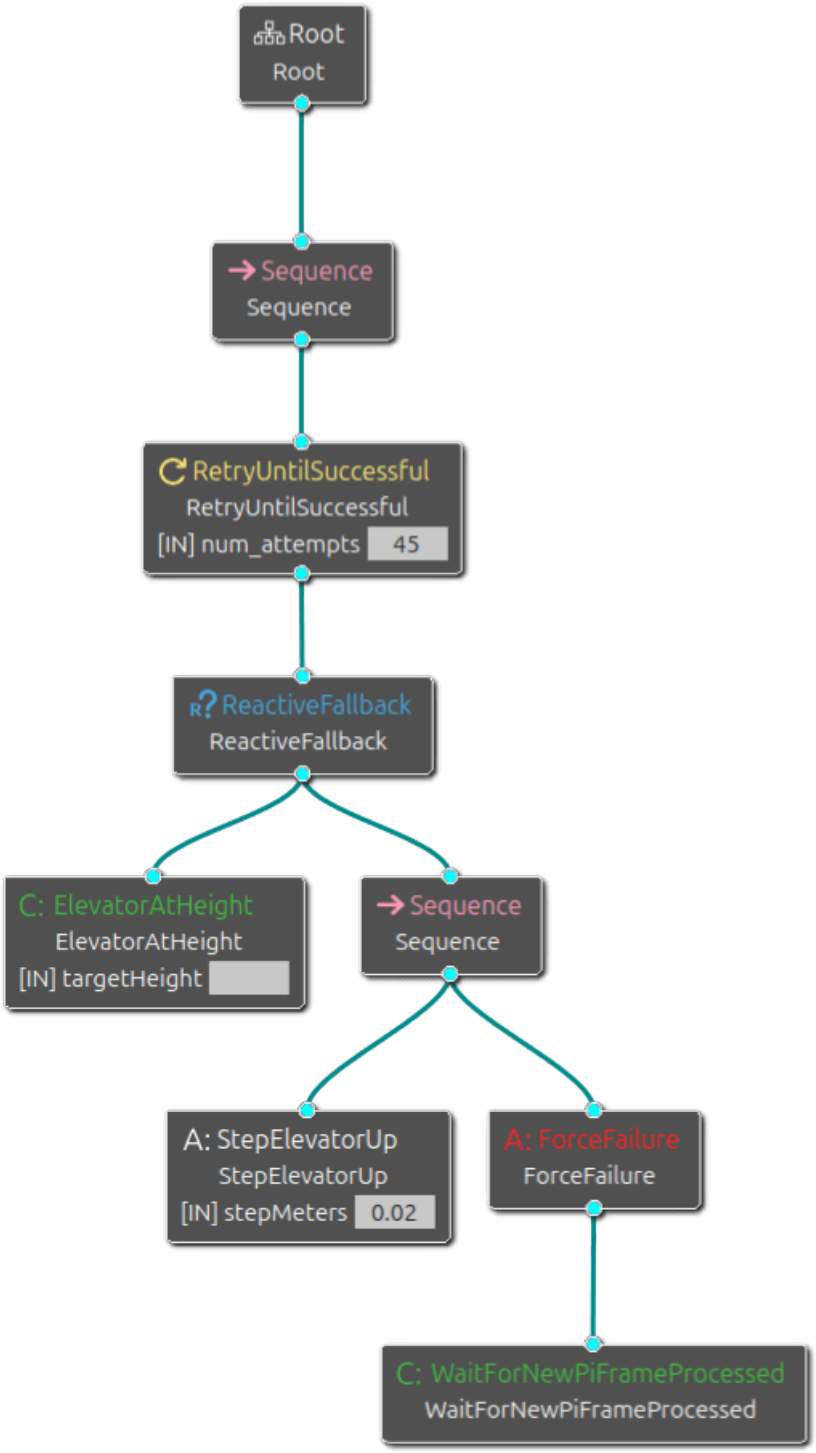

Figure 8: RaiseElevatorToCanHeight

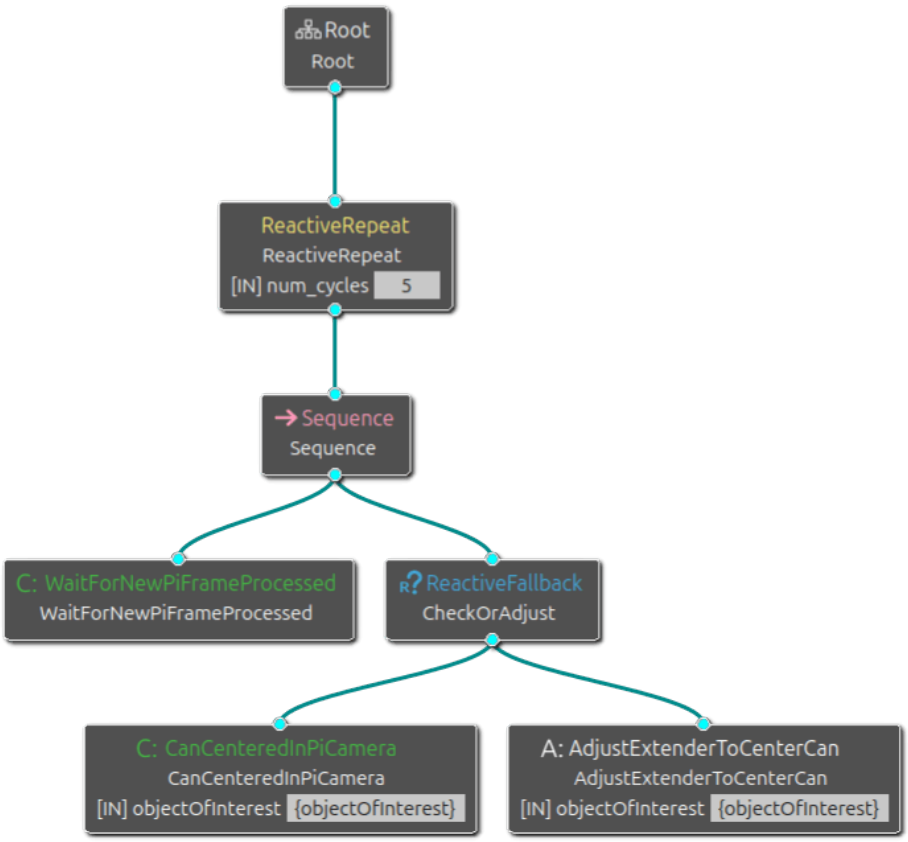

Figure 9: CenterCanInPiCamera

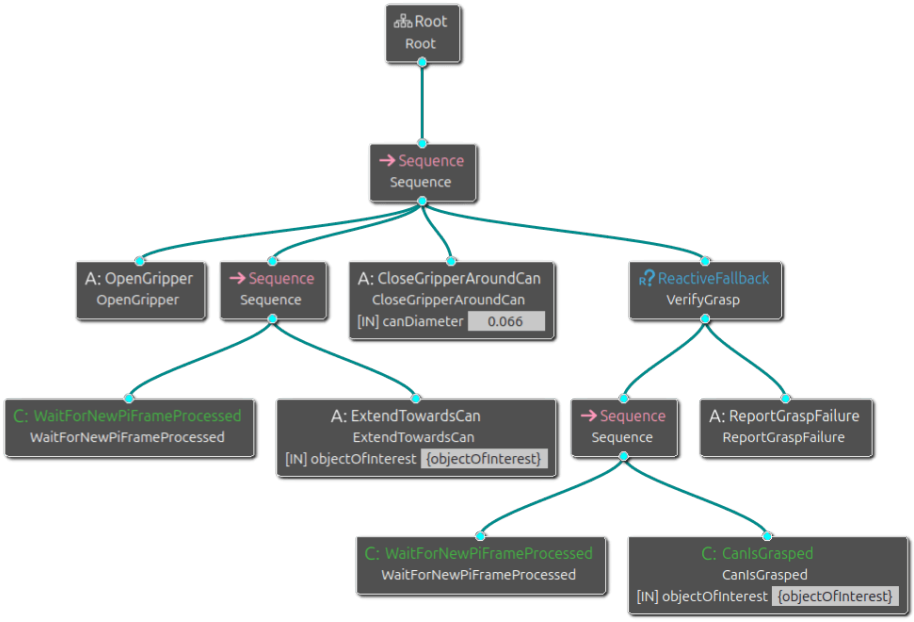

Figure 10: ExecuteGrasp

Returning Home

The mission cleanup is handled by the root sequence. `BackAwayFromTable` simply reverses the wheels for 0.3 meters to ensure we don't clip the table when turning.

Finally, `ReturnToStart` uses the pose we captured at the very beginning of the tree (`SaveRobotPose`). It sends a standard Nav2 action to bring the beer back to where we started.

Critique and Future Work

This behavior tree was designed for a specific "happy path" scenario: the Can-Do Challenge. While it works for the specific constraints of the competition, there are several engineering "shortcuts" that a production robot would need to address.

Current limitations

1. **Limited Recoveries**: If the can slips out of the gripper during the lift, `VerifyGrasp` reports failure, but there is no higher-level loop to re-attempt the grasp or search for the can on the floor. The robot essentially gives up.
2. **Blind Extension**: The `ExtendTowardsCan` node trusts the vision system implicitly. There is no force feedback or tactile checking. If the distance

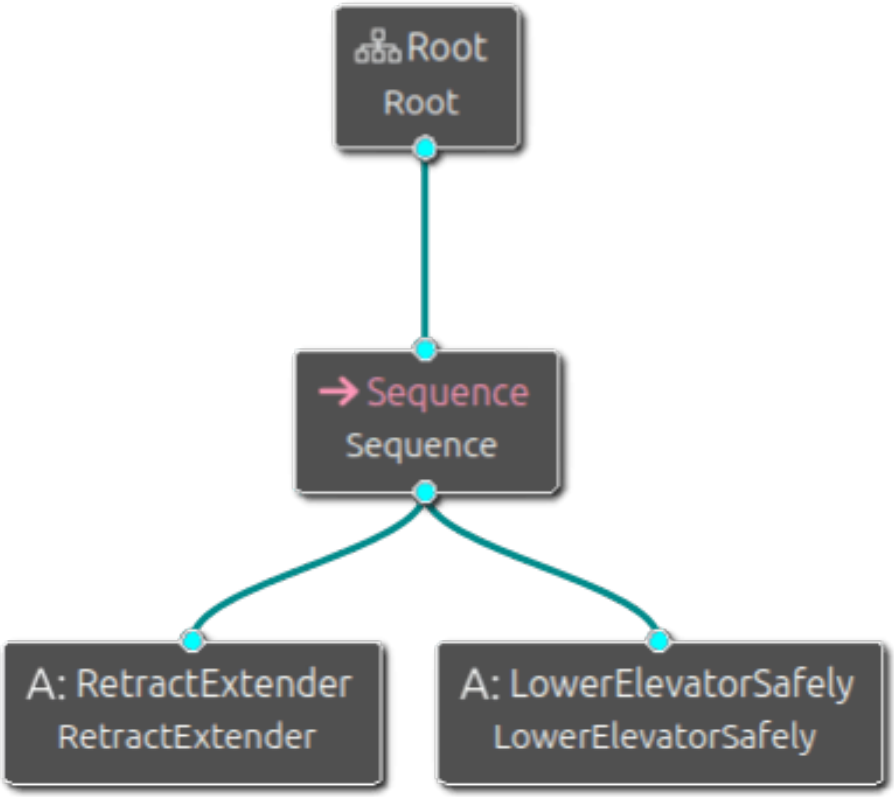

Figure 11: RetractAfterGrasp

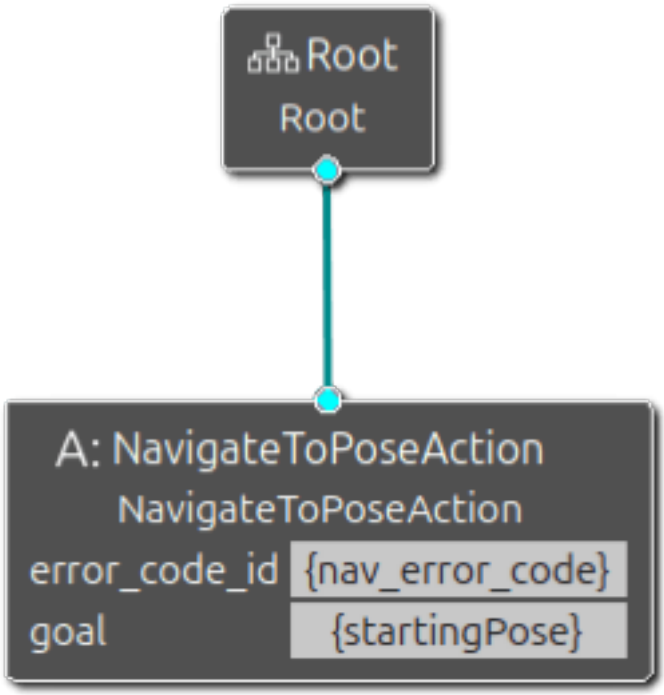

Figure 12: ReturnToStart

estimate is off by 5cm, we might tip the can over or stalemate the arm against the table.

3. **Single-Pass Navigation**: The `NavigateToCanLocation` happens once. If the robot arrives and doesn't see the can (perhaps it's occluded), it doesn't try to move to a new vantage point.
4. **Field-of-View Dependency**: The hand-off between OAK-D (remote approach) and Pi Camera (manipulation) is brittle. If the robot approaches too close or at a bad angle, the can might drop out of the Pi Camera's view, causing the `ElevatorAtHeight` check to fail indefinitely.
5. **Weak Approach Pose Selection**: Once the robot is in the neighborhood, it drives straight toward the can. That might be a poor approach pose if the can is far from the table edge. A slight arc or a position adjustment could set up a better grasp.
6. **No Active Search Strategy**: The expected can pose gets us close, but if the can isn't visible from that pose, the robot doesn't systematically search nearby viewpoints. Rotation helps, but it's not enough.

Next version ideas

- **Viewpoint search**: Add a small pattern of offsets (arc, lateral step, or circle) around the expected pose to improve visibility.
- **Approach pose optimization**: Evaluate multiple approach headings to keep the can within a comfortable reach envelope.
- **Recovery loops**: If the grasp fails, back out, re-acquire the can, and try again.

However, as a proof of concept for **reactive planning**, it demonstrates the power of Behavior Trees: safety is never compromised, perception drives action directly, and the state machine logic is visible and modular. Not bad for a beer run.