https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

https://docs.nav2.org/behavior_trees/overview/detailed_behavior_tree_walkthrough.html

https://docs.nav2.org/behavior_trees/trees/follow_point.html

https://docs.nav2.org/behavior_trees/trees/nav_to_pose_with_consistent_replanning_and_if_path_becomes_invalid.html

https://docs.nav2.org/behavior_trees/trees/nav_to_pose_and_pause_near_goal_obstacle.html

Introduction to BTs

behavior Tree is a tree of hierarchical nodes that controls the flow of execution of "tasks".

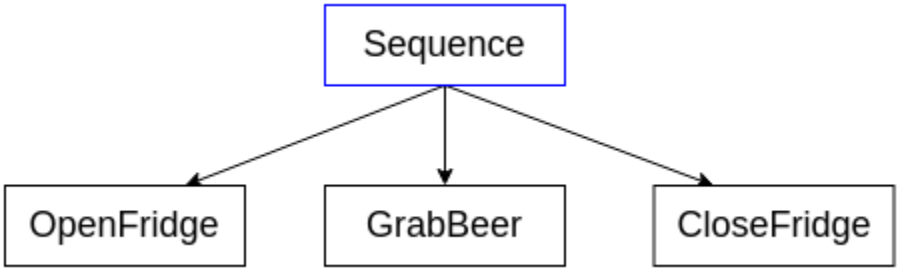
- A signal called "tick" is sent to the root of the tree and propagates through the tree until it reaches a leaf node.
- Any TreeNode that receives a tick signal executes its callback. This callback must return either
    - SUCCESS
    - FAILURE
    - RUNNING
- RUNNING means that the action needs more time to return a valid result.
- The LeafNodes, those TreeNodes which don't have any children, are the actual commands. Action nodes are the most common type of LeafNodes.
  https://www.behaviortree.dev/docs/learn-the-basics/BT_basics
  https://docs.nav2.org/configuration/packages/configuring-bt-xml.html

Type of TreeNode	Children Count	Notes
ControlNode	1...N	Usually, ticks a child based on the result of its siblings or/and its own state.

DecoratorNode	1	Among other things, it may alter the result of its child or tick it multiple times.
ConditionNode	0	Should not alter the system. Shall not return RUNNING.
ActionNode	0	This is the Node that "does something".

ControlNode

Sequence ->> (AND) logic



- If a child returns SUCCESS, tick the next one.
- If a child returns FAILURE, then no more children are ticked, and the Sequence returns FAILURE.
- If all the children return SUCCESS, then the Sequence returns SUCCESS too.

PipelineSequence >> (Asynchronous execution)

- Ticks the first child till it succeeds,
- then ticks the first and second children till the second one succeeds.
- It then ticks the first, second, and third children until the third succeeds, and so on, and so on.
- If at any time a child returns RUNNING, that doesn't change the behavior.

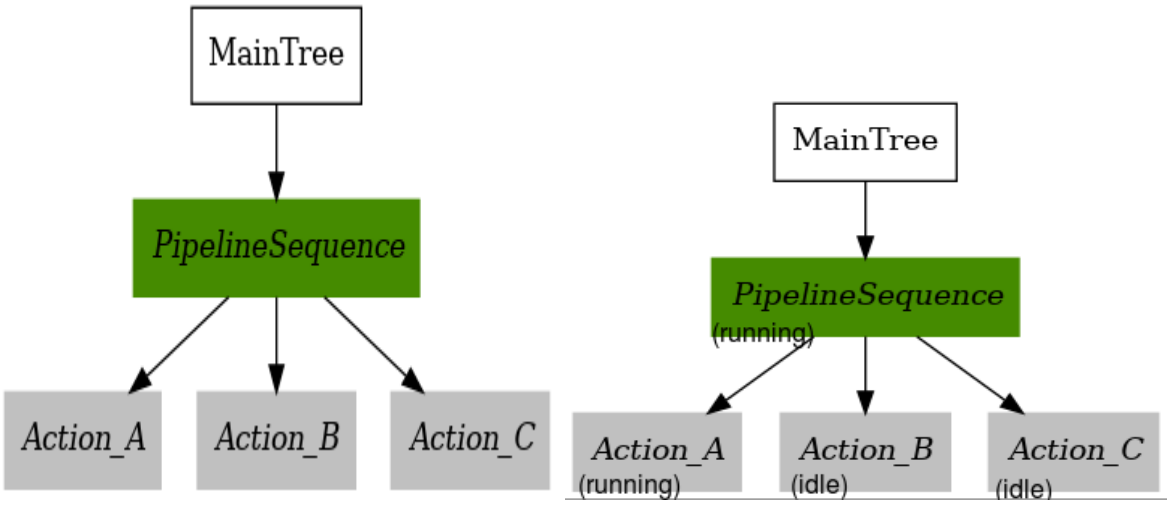
- If at any time a child returns FAILURE, that stops all children and returns FAILURE overall.

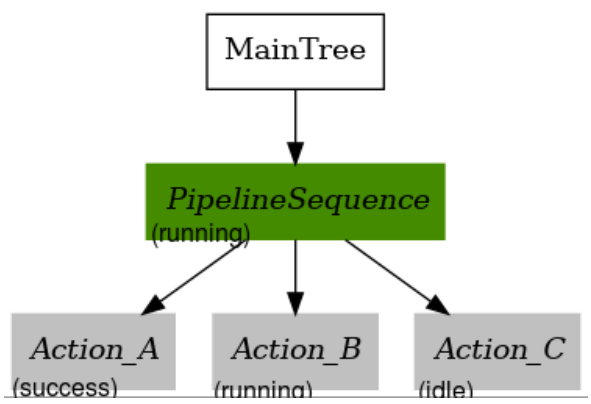
```
<PipelineSequence>
    <!--Add tree components here--->
</PipelineSequence>
```

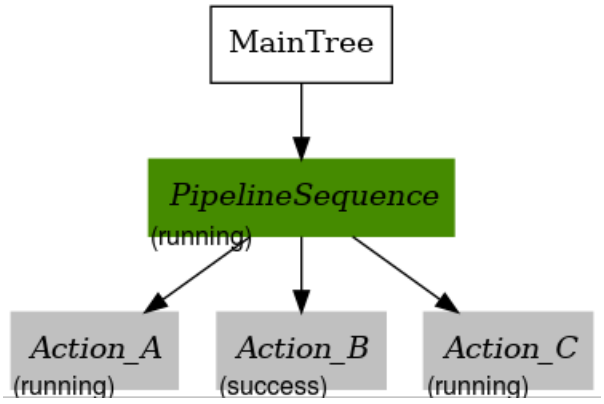
https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

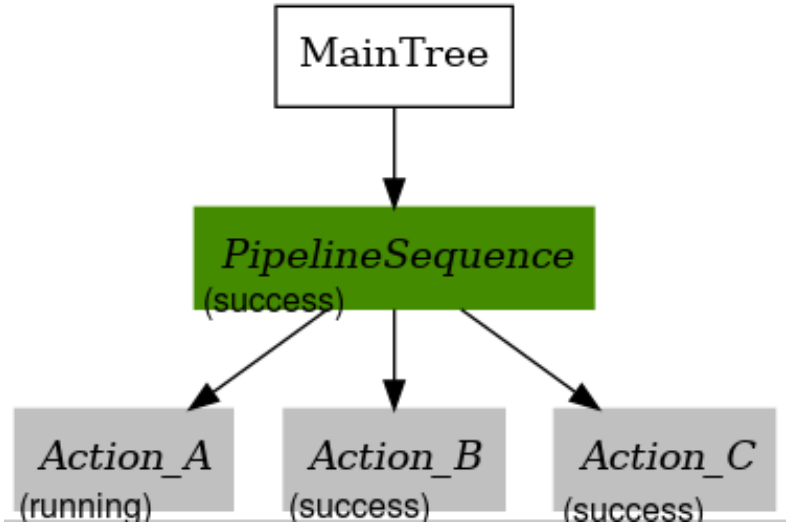
To explain this further, here is an example BT that uses PipelineSequence.

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```









A, AB, ABC (tick step)

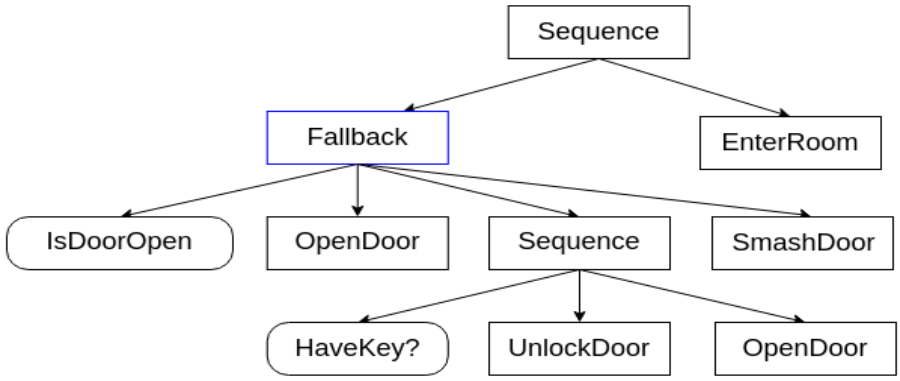
A, AB, ABC, …., ABC(N) ← pattern

1. Action_A, Action_B, and Action_C are all IDLE.
2. When the parent PipelineSequence is first ticked, let's assume Action_A returns RUNNING. The parent node will now return RUNNING and no other nodes are ticked.
3. Now, let's assume Action_A returns SUCCESS,

Action_B will now get ticked and will return RUNNING. Action_C has not yet been ticked so will return IDLE.

4. Action_A gets ticked again and returns RUNNING, and Action_B gets re-ticked and returns SUCCESS and therefore the BT goes on to tick Action_C for the first time. Let's assume Action_C returns RUNNING. The retick-ing of Action_A is what makes PipelineSequence useful.
5. All actions in the sequence will be re-ticked. Let's assume Action_A still returns RUNNING, where as Action_B returns SUCCESS again, and Action_C now returns SUCCESS on this tick. The sequence is now complete, and therefore Action_A is halted, even though it was still RUNNING.
6. Recall that if `Action_A`, `Action_B`, or `Action_C` returned `FAILURE` at any point of time, the parent would have returned `FAILURE` and halted any children as well.

Fallback ->> (OR) logic

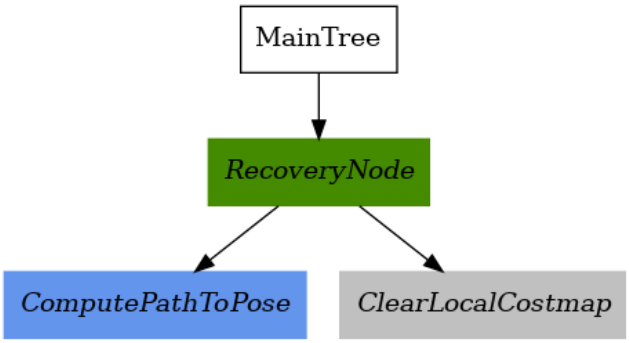


- If a child returns FAILURE, tick the next one.
- If a child returns SUCCESS, then no more children are ticked and the Fallback returns SUCCESS.
- If all the children return FAILURE, then the Fallback returns FAILURE too.

Recovery >>

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RecoveryNode number_of_retries="1">
            <ComputePathToPose/>
```

```
            <ClearLocalCostmap/>
        </RecoveryNode>
    </BehaviorTree>
</root>
```



The Recovery control node has only two children and returns `SUCCESS` if and only if the first child returns `SUCCESS`. If the first child returns `FAILURE`, the second child will be ticked. This loop will continue until either:

- The first child returns `SUCCESS` (which results in `SUCCESS` of the parent node)
- The second child returns `FAILURE` (which results in `FAILURE` of the parent node)
- The `number_of_retries` input parameter is violated

In the above example, let's assume `ComputePathToPose` fails. `ClearLocalCostmap` will be ticked in response, and return `SUCCESS`. Now that we have cleared the costmap, let's say the robot is correctly able to compute the path and `ComputePathToPose` now returns `SUCCESS`. Then, the parent RecoveryNode will also return `SUCCESS` and the BT will be complete.

Number_of_retries —>

```
<RecoveryNode number_of_retries="1">

    <!--Add tree components here--->

</RecoveryNode>
```

Type	Default
int	1

In nav2, the RecoveryNode is included in Behavior Trees to implement recovery actions upon failures.

https://www.behaviortree.dev/docs/tutorial-basics/tutorial_04_sequence/

Reactive and Asynchronous behaviors

Sequence >>

when `executeTick()` was called, `MoveBase` returned RUNNING the 1st and 2nd time, and eventually SUCCESS the 3rd time.

`BatteryOK` is executed only once.

```
<root BTCPP_format="4">
    <BehaviorTree>
        <Sequence>
            <BatteryOK/>
            <SaySomething   message="mission started..." />
            <MoveBase           goal="1;2;3"/>
            <SaySomething   message="mission completed!" />
        </Sequence>
    </BehaviorTree>
</root>
```

```
Expected output:
```

```
--- ticking
[ Battery: OK ]
Robot says: mission started...
[ MoveBase: SEND REQUEST ]. goal: x=1.0 y=2.0 theta=3.0
--- status: RUNNING

--- ticking
--- status: RUNNING

--- ticking
[ MoveBase: FINISHED ]
Robot says: mission completed!
--- status: SUCCESS
```

Reactive Sequence >>

If we use a `ReactiveSequence`, when the child `MoveBase` returns RUNNING, the sequence is restarted and the condition `BatteryOK` is executed again.

```
<root>
    <BehaviorTree>
       <ReactiveSequence>
           <BatteryOK/>
           <Sequence>
               <SaySomething   message="mission started..." />
               <MoveBase           goal="1;2;3"/>
               <SaySomething   message="mission completed!" />
           </Sequence>
       </ReactiveSequence>
    </BehaviorTree>
</root>
```

```
Expected output:
--- ticking
[ Battery: OK ]
Robot says: mission started...
[ MoveBase: SEND REQUEST ]. goal: x=1.0 y=2.0 theta=3.0
--- status: RUNNING
```

```
--- ticking
[ Battery: OK ]
--- status: RUNNING

--- ticking
[ Battery: OK ]
[ MoveBase: FINISHED ]
Robot says: mission completed!
--- status: SUCCESS
```

```
<ReactiveSequence>
     <Inverter>
        <PathExpiringTimer seconds="10" path="{path}"/>
     </Inverter>
     <Inverter>
         <GlobalUpdatedGoal/>
      </Inverter>
      <IsPathValid path="{path}"/>
</ReactiveSequence>
```

- Scenario: A robot navigation system continuously monitors the validity of its path while reacting to changes in goals or path expiration.
- Behavior: The robot will stop and re-evaluate its path if:
    - The path is about to expire in 10 seconds.
    - There is an updated global goal.
    - The current path is no longer valid.

Example 1: Sequence (Sequencer)

Scenario: A robot is performing a series of tasks to clean a room.

1. Move to Room: The robot moves to the designated room.
2. Identify Trash: The robot scans the room to identify trash.
3. Pick Up Trash: The robot picks up the identified trash.
4. Move to Trash Bin: The robot moves to the trash bin.

5. Dispose Trash: The robot disposes of the trash.

In this Sequence, each step must be completed successfully before moving on to the next step. If any step fails (e.g., if the robot cannot identify trash), the Sequence fails and stops executing further steps.

Example 2: Reactive Sequence

Scenario: A security system continuously monitors the status of a building.

1. Check Door Sensor: Continuously checks if any door is open.
2. Check Window Sensor: Continuously checks if any window is open.
3. Check Motion Sensor: Continuously checks if there is any motion detected inside the building.
4. Alert System: If any of the sensors are triggered, the system raises an alert.

In this Reactive Sequence, the system continuously reevaluates the state of each sensor. It does not stop after one check but keeps monitoring the sensors in every tick cycle. If any sensor is triggered (e.g., a door is opened), the Reactive Sequence immediately reacts and triggers the alert system.

ReactiveFallback

Concept:

- A ReactiveFallback is similar to a Fallback, but it continuously reevaluates its children each tick cycle, allowing for dynamic and responsive behavior.
- It doesn't stop after a single success; instead, it continuously checks all children to ensure the best possible outcome.

```
<ReactiveFallback>

   <CheckDoorSensor/>

   <CheckWindowSensor/>

   <CheckMotionSensor/>

   <AlertSystem/>

</ReactiveFallback>
```

Fallback

Concept:

- A Fallback (also known as Selector) node is designed to attempt multiple actions in sequence until one succeeds.
- If one child node returns success, the Fallback node returns success immediately.
- If all child nodes return failure, the Fallback node returns failure.

```
<Fallback>

   <PrimaryPath/>

   <SecondaryPath/>

   <TertiaryPath/>

</Fallback>
```

Aspect	Fallback (Selector)	ReactiveFallback
Evaluation	Evaluates children in sequence until one succeeds or all fail.	Continuously reevaluates all children each tick cycle.
Behavior	Stops evaluating after the first success.	Keeps monitoring and evaluating all children continuously.

Use Case	Suitable for scenarios where a single success path is sufficient.	Suitable for dynamic and responsive scenarios where continuous monitoring is needed.
Example	Robot trying multiple paths to reach a destination.	Home security system continuously monitoring for intrusions.

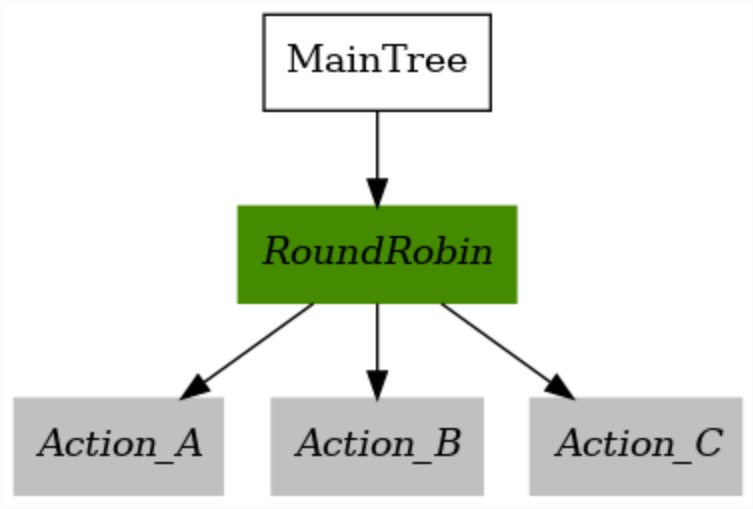
Aspect	**Sequence (Sequencer)**	**Reactive Sequence**
Execution Order	Ticks children nodes in a predefined order until one fails or all succeed.	Continuously ticks children nodes in a predefined order each tick cycle.
Success Criteria	Succeeds if all children succeed. If one fails, it stops and returns failure.	Returns success if all children return success. Continuously reevaluates children.

Failure Criteria	Fails if any child fails.	Fails if any child fails. Continuously reevaluates children.
Use Case	Used when a sequence of actions needs to be executed in a specific order.	Used for constantly checking conditions or actions that need to be reevaluated frequently.
State Management	Does not reevaluate children once a child fails or succeeds in a tick cycle.	Continuously reevaluates children, useful for dynamic and reactive behaviors.
Example Scenario	A sequence of tasks that must be completed in a specific order, such as a robot navigating through waypoints.	Monitoring sensors and reacting to their state changes continuously.

RoundRobin

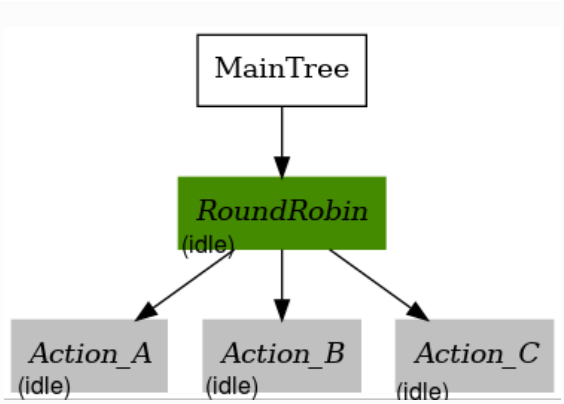
The RoundRobin control node ticks its children in a round robin fashion until a child returns `SUCCESS`, in which the parent node will also return `SUCCESS`. If all children return `FAILURE` so will the parent RoundRobin.

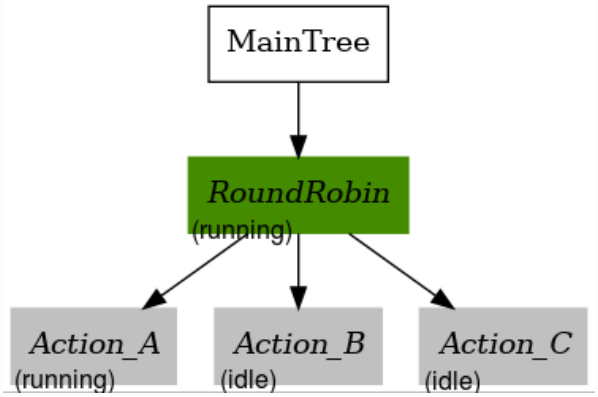
https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

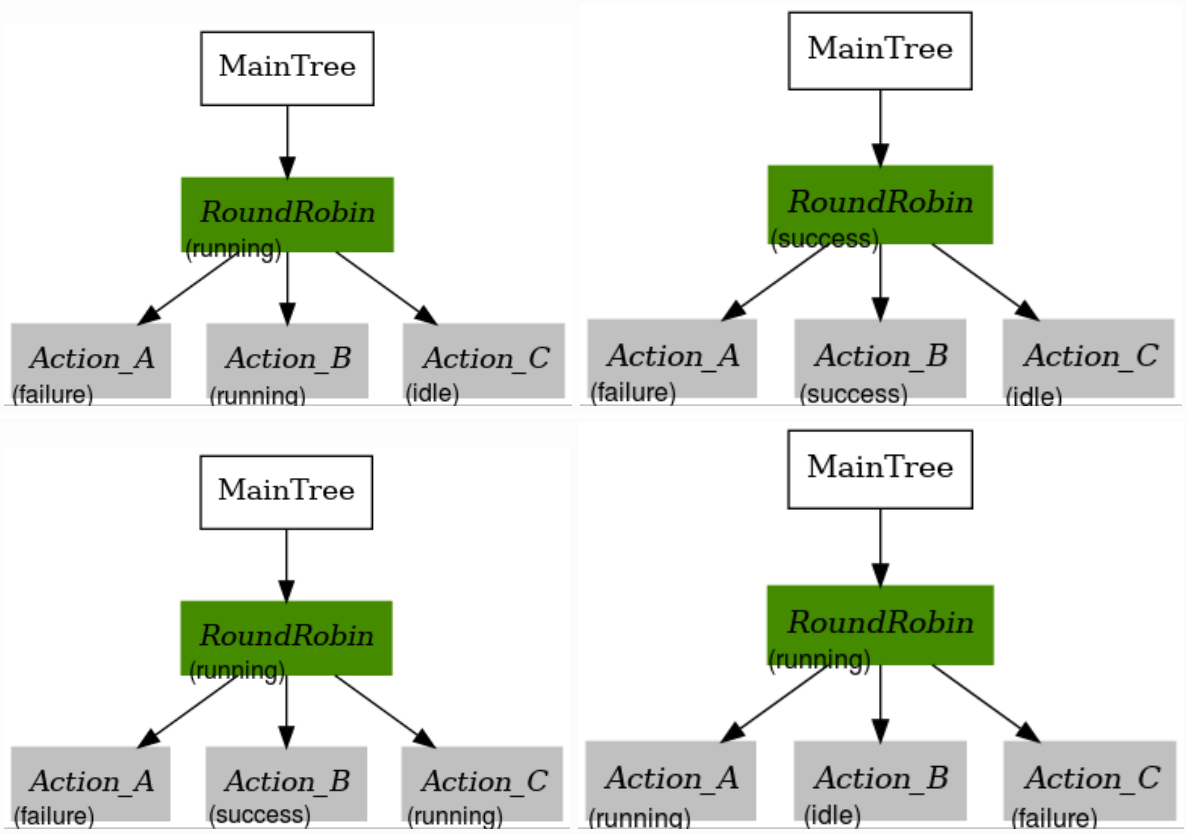



```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```

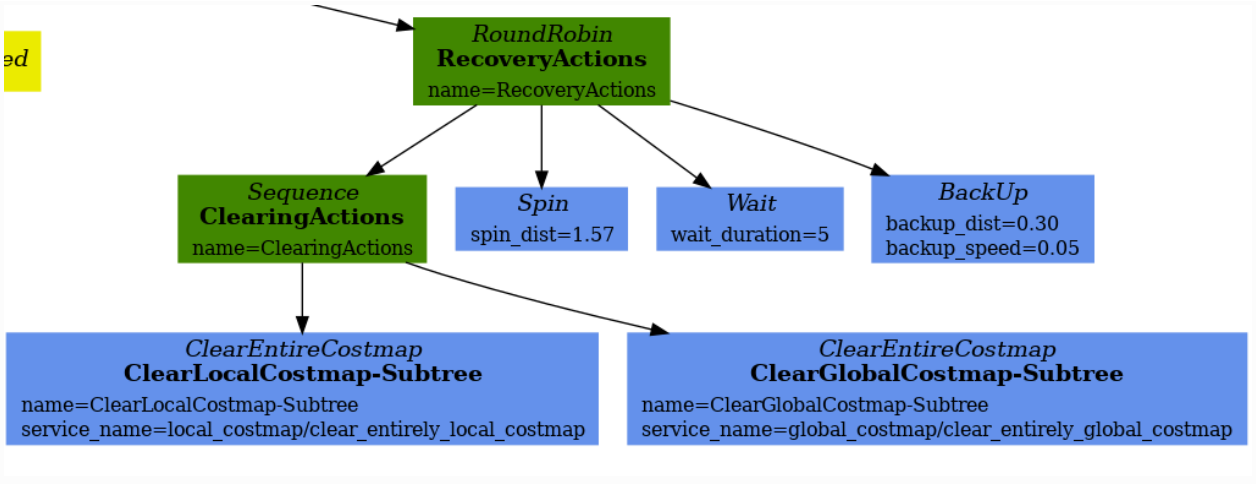
1. All the nodes start at `IDLE`







Behavior:




```
<RoundRobin name="RecoveryActions">
     <Sequence name="ClearingActions">
         <ClearEntireCostmap name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_entirely_local_costmap"/>
          <ClearEntireCostmap name="ClearGlobalCostmap-Subtree"
service_name="global_costmap/clear_entirely_global_costmap"/>
       </Sequence>
```

```
        <Spin spin_dist="1.57"/>
         <Wait wait_duration="5"/>
         <BackUp backup_dist="0.15" backup_speed="0.025"/>
</RoundRobin>
```

- RoundRobin Execution:
  - First Tick:
    - Executes the `ClearingActions` sequence.
    - Clears the local and global costmaps in order.
    - If both clearing actions succeed, the sequence returns success; otherwise, it fails.
  - Second Tick:
    - Executes the `Spin` action.
    - The robot spins in place.
  - Third Tick:
    - Executes the `Wait` action.
    - The robot waits for 5 seconds.
  - Fourth Tick:
    - Executes the `BackUp` action.
    - The robot backs up for 0.30 meters.

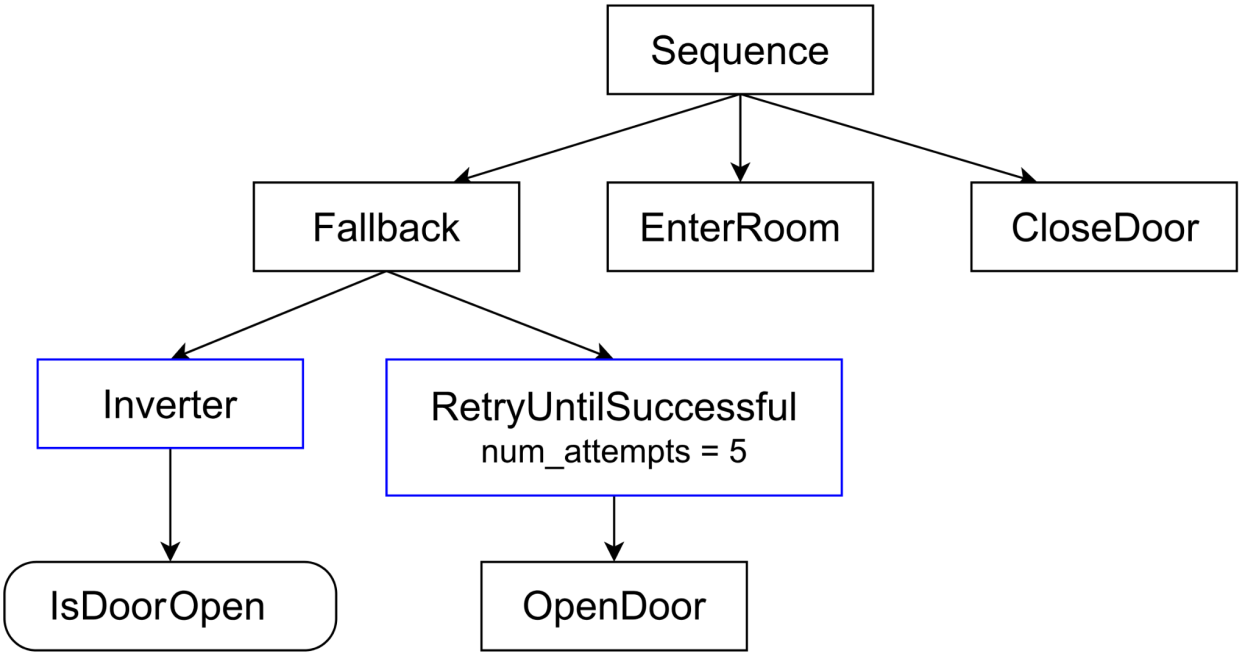
Round Robin with reactive fall back >>

```
<ReactiveFallback name="RecoveryFallback">
       <GoalUpdated/>
       <RoundRobin name="RecoveryActions">
         <Sequence name="ClearingActions">
           <ClearEntireCostmap name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_entirely_local_costmap"/>
           <ClearEntireCostmap name="ClearGlobalCostmap-Subtree"
service_name="global_costmap/clear_entirely_global_costmap"/>
         </Sequence>
         <Spin spin_dist="1.57"/>
         <Wait wait_duration="5"/>
         <BackUp backup_dist="0.30" backup_speed="0.05"/>
       </RoundRobin>
```

```
</ReactiveFallback>
```

DecoratorNode (Plugin)

- to transform the result it received from the child.
- to halt the execution of the child.
- to repeat ticking the child, depending on the type of Decorator.



The node Inverter is a Decorator that inverts the result returned by its child; An Inverter followed by the node called isDoorOpen is, therefore, equivalent to
"Is the door closed?".
The node Retry will repeat ticking the child up to num_attempts times (5 in this case) if the child returns FAILURE.
Apparently, the branch on the left side means:
If the door is closed, then try to open it.
Try up to 5 times, otherwise give up and return FAILURE.

- RateController

A node that throttles the tick rate for its child. The tick rate can be supplied to the node as a parameter. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `RateController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

```
<RateController hz="1.0">
    <!--Add tree components here--->
</RateController>
```

- DistanceController

A node that controls the tick rate for its child based on the distance traveled. The distance to be traveled before replanning can be supplied to the node as a parameter. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `DistanceController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

```
<DistanceController distance="0.5" global_frame="map"
robot_base_frame="base_link">
  <!--Add tree components here--->
</DistanceController>
```

- SpeedController

A node that controls the tick rate for its child based on current robot speed. The maximum and minimum replanning rates can be supplied to the node as parameters along with maximum and minimum speed. The node returns RUNNING when it is not ticking its child. Currently, in the navigation stack, the `SpeedController` is used to adjust the rate at which the `ComputePathToPose` and `GoalReached` nodes are ticked.

```
<SpeedController min_rate="0.1" max_rate="1.0" min_speed="0.0"
max_speed="0.5" filter_duration="0.3">
  <!--Add tree components here--->
</SpeedController>
```

- GoalUpdater

  A custom control node, which updates the goal pose. It subscribes to a topic in which it can receive an updated goal pose to use instead of the one commanded in action. It is useful for dynamic object following tasks.

  ```
  <GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
    <!--Add tree components here--->
  </GoalUpdater>
  ```

- PathLongerOnApproach

  This node checks if the newly generated global path is significantly larger than the old global path in the user-defined robot's goal proximity and triggers their corresponding children. This allows users to enact special behaviors before a robot attempts to execute a path significantly longer than the prior path when close to a goal (e.g. going around an dynamic obstacle that may just need a few seconds to move out of the way).

  ```
  <PathLongerOnApproach path="{path}" prox_len="3.0" length_factor="2.0">
    <!--Add tree components here--->
  </PathLongerOnApproach>
  ```

- SingleTrigger

  This node triggers its child only once and returns FAILURE for every succeeding tick.

  ```
  <SingleTrigger>
    <!--Add tree components here--->
  </SingleTrigger>
  ```

- Inverter

Invert the output False→ True and True→ False

```
<Inverter>
  <PathExpiringTimer seconds="10" path="{path}"/>
</Inverter>
<Inverter>
  <GlobalUpdatedGoal/>
</Inverter>
<IsPathValid path="{path}"/>
```

ConditionNode

- GoalReached

Checks the distance to the goal, if the distance to goal is less than the pre-defined threshold, the tree returns SUCCESS, otherwise it returns FAILURE.
bt_navigator:
  ros__parameters:
    transform_tolerance: 0.1
    goal_reached_tol: 0.25

```
<GoalReached goal="{goal}" robot_base_frame="base_link"/>
```

- TransformAvailable

Checks if a TF transform is available. Returns failure if it cannot be found. Once found, it will always return success. Useful for initial condition checks.

```
<TransformAvailable parent="odom" child="base_link"/>
```

- DistanceTraveled

Node that returns success when a configurable distance has been traveled.

```
bt_navigator:
  ros__parameters:
    # other bt_navigator parameters
    transform_tolerance: 0.1
```

```
<DistanceTraveled distance="0.8" global_frame="map" robot_base_frame="base_link"/>
```

- GoalUpdated

Checks if the global navigation goal, or a vector of goals, has changed in the blackboard. Returns failure if the goal is the same, if it changes, it returns success.

```
<GoalUpdated/>
```

- GloballyUpdatedGoal

Checks if the global navigation goal has changed in the blackboard. Returns failure if the goal is the same, if it changes, it returns success.

This node differs from the GoalUpdated by retaining the state of the current goal/goals throughout each tick of the BehaviorTree such that it will update on any “global” change to the goal.

```
<GlobalUpdatedGoal/>
```

- InitialPoseReceived

  Node that returns success when the initial pose is sent to AMCL via /initial_pose

  ```
  <InitialPoseReceived/>
  ```

- IsStuck

Determines if the robot is not progressing towards the goal. If the robot is stuck and not progressing, the condition returns SUCCESS, otherwise it returns FAILURE.

```
<IsStuck/>
```

- TimeExpired

  Node that returns success when a time duration has passed

  ```
  <TimeExpired seconds="1.0"/>
  ```

- IsBatteryLow

  Checks if battery is low by subscribing to a `sensor_msgs/BatteryState` topic and checking if battery percentage/voltage is below a specified minimum value. By

default percentage (in range 0 to 1) is used to check for low battery. Set the `is_voltage` parameter to true to use voltage. Returns SUCCESS when battery percentage/voltage is lower than the specified value, FAILURE otherwise.

```
<IsBatteryLow min_battery="0.5" battery_topic="/battery_status"
is_voltage="false"/>
```

- IsPathValid

Checks to see if the global path is valid. If there is an obstacle along the path, the condition returns FAILURE, otherwise it returns SUCCESS.

```
<IsPathValid server_timeout="10" path="{path}"/>
```

- PathExpiringTimer

Check if the timer has expired. Returns success if the timer has expired, otherwise it returns failure. The timer will reset if the path gets updated.

```
<PathExpiringTimer seconds="15" path="{path}"/>
```

- AreErrorCodesPresent

Checks the if the provided error code matches any error code within a set.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

```
<AreErrorCodesPresent error_code="{error_code}"
error_codes_to_check="{error_codes_to_check}"/>
<AreErrorCodesPresent error_code="{error_code}"
error_codes_to_check="101,107,119"/>
```

- WouldAControllerRecoveryHelp

  Checks if the active controller server error code is UNKNOWN, PATIENCE_EXCEEDED, FAILED_TO_MAKE_PROGRESS, or NO_VALID_CONTROL.

  If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

  ```
  <WouldAControllerRecoveryHelp error_code="{follow_path_error_code}"/>
  ```

- WouldAPlannerRecoveryHelp

  Checks if the active controller server error code is UNKNOWN, NO_VALID_CONTROL, or TIMEOUT.

  If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

  ```
  <WouldAPlannerRecoveryHelp error_code="{compute_path_to_pose_error_code}"/>
  ```

- WouldASmootherRecoveryHelp

  Checks if the active controller server error code is UNKNOWN, TIMEOUT, FAILED_TO_SMOOTH_PATH, or SMOOTHED_PATH_IN_COLLISION.

If the active error code is a match, the node returns `SUCCESS`. Otherwise, it returns `FAILURE`.

```
<WouldASmootherRecoveryHelp error_code="{smoother_error_code}"/>
```

- IsBatteryCharging

Checks if the battery is charging by subscribing to a `sensor_msgs/BatteryState` topic and checking if the power_supply_status is `POWER_SUPPLY_STATUS_CHARGING`. Returns SUCCESS in that case, FAILURE otherwise.

```
<IsBatteryCharging battery_topic="/battery_status"/>
```

Action Plugins

- Wait

Invokes the Wait ROS 2 action server, which is implemented by the nav2_behaviors module. This action is used in nav2 Behavior Trees as a recovery behavior.

```
<Wait wait_duration="1.0" server_name="wait_server" server_timeout="10"/>
```

- Spin

Invokes the Spin ROS 2 action server, which is implemented by the nav2_behaviors module. It performs an in-place rotation by a given angle. This action is used in nav2 Behavior Trees as a recovery behavior.

```
<Spin spin_dist="1.57" server_name="spin" server_timeout="10"
is_recovery="true" error_code_id="{spin_error_code}"/>
```

- BackUp

Invokes the BackUp ROS 2 action server, which causes the robot to back up by a specific displacement. It performs an linear translation by a given distance. This is used in nav2 Behavior Trees as a recovery behavior. The nav2_behaviors module implements the BackUp action server.

```
<BackUp backup_dist="-0.2" backup_speed="0.05" server_name="backup_server"
server_timeout="10" error_code_id="{backup_error_code}"/>
```

- DriveOnHeading

Invokes the DriveOnHeading ROS 2 action server, which causes the robot to drive on the current heading by a specific displacement. It performs a linear translation by a given distance. The nav2_behaviors module implements the DriveOnHeading action server.

```
<DriveOnHeading dist_to_travel="0.2" speed="0.05" server_name="backup_server"
server_timeout="10" error_code_id="{drive_on_heading_error_code}"/>
```

- AssistedTeleop

Invokes the AssistedTeleop ROS 2 action server, which filters teleop twist commands to prevent collisions. This is used in nav2 Behavior Trees as a recovery behavior or a regular behavior. The nav2_behaviors module implements the AssistedTeleop action server.

```
<AssistedTeleop is_recovery="false" server_name="assisted_teleop_server"
server_timeout="10" error_code_id="{assisted_teleop_error_code}"/>
```

- ComputePathToPose

Invokes the ComputePathToPose ROS 2 action server, which is implemented by the nav2_planner module. The server address can be remapped using the server_name input port.

```
<ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"
server_name="ComputePathToPose" server_timeout="10"
error_code_id="{compute_path_error_code}"/>
```

- FollowPath

Invokes the FollowPath ROS 2 action server, which is implemented by the controller plugin modules loaded. The server address can be remapped using the server_name input port.

```
FollowPath path="{path}" controller_id="FollowPath"
goal_checker_id="precise_goal_checker" server_name="FollowPath"
server_timeout="10" error_code_id="{follow_path_error_code}"/>
```

- NavigateToPose

Invokes the NavigateToPose ROS 2 action server, which is implemented by the bt_navigator module.

```
<NavigateToPose goal="{goal}" server_name="NavigateToPose" server_timeout="10"
error_code_id="{navigate_to_pose_error_code}"
behavior_tree="<some-path>/behavior_trees/navigate_through_poses_w_replanning_a
nd_recovery.xml"/>
```

- ClearEntireCostmap

Action to call a costmap clearing server.

```
<ClearEntireCostmap name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_entirely_local_costmap"/>
```

```
<ClearEntireCostmap name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_entirely_local_costmap"/>


          <ClearEntireCostmap name="ClearGlobalCostmap-Subtree"
service_name="global_costmap/clear_entirely_global_costmap"/>
```

- ClearCostmapExceptRegion

Action to call a costmap clearing except region server

```
<ClearCostmapExceptRegion name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_except_local_costmap"/>
```

- ClearCostmapAroundRobot

Action to call a costmap clearing around the robot server.

```
      <ClearCostmapAroundRobot name="ClearLocalCostmap-Subtree"
service_name="local_costmap/clear_around_local_costmap"/>
```

- ReinitializeGlobalLocalization

Used to trigger global relocalization using AMCL in case of severe delocalization or kidnapped robot problem.

```
<ReinitializeGlobalLocalization
service_name="reinitialize_global_localization"/>
```

- TruncatePath

A custom control node, which modifies a path making it shorter. It removes parts of the path closer than a distance to the goal pose. The resulting last pose of the path orientates the robot to the original goal pose.

```
<TruncatePath distance="1.0" input_path="{path}"
output_path="{truncated_path}"/>
```

- TruncatePathLocal

A custom control node, which modifies a path making it shorter. It removes parts of the path which are more distant than specified forward/backward distance around robot.

```
<TruncatePathLocal input_path="{path}" output_path="{path_local}"
distance_forward="3.5" distance_backward="2.0" robot_frame="base_link"/>
```

- PlannerSelector

It is used to select the planner that will be used by the planner server. It subscribes to the planner_selector topic to receive command messages with the name of the planner to be used. It is commonly used before the ComputePathToPoseAction. The selected_planner output port is passed to the planner_id input port of the ComputePathToPoseAction. If none is provided on the topic, the default_planner is used.
Any publisher to this topic needs to be configured with some QoS defined as reliable and transient local.

```
<PlannerSelector selected_planner="{selected_planner}"
default_planner="GridBased" topic_name="planner_selector"/>
```

- ControllerSelector

It is used to select the Controller that will be used by the Controller server. It subscribes to the controller_selector topic to receive command messages with the name of the Controller to be used. It is commonly used before of the FollowPathAction. The selected_controller output port is passed to controller_id input port of the FollowPathAction. If none is provided on the topic, the default_controller is used.
Any publisher to this topic needs to be configured with some QoS defined as reliable and transient local.

```
<ControllerSelector selected_controller="{selected_controller}"
default_controller="FollowPath" topic_name="controller_selector"/>
```

- SmootherSelector

It is used to select the Smoother that will be used by the Smoother server. It subscribes to the smoother_selector topic to receive command messages with the name of the Smoother to be used. It is commonly used before the FollowPathAction. If none is provided on the topic, the default_smoother is used.
Any publisher to this topic needs to be configured with some QoS defined as reliable and transient local.

```
<SmootherSelector selected_smoother="{selected_smoother}"
default_smoother="SimpleSmoother" topic_name="smoother_selector"/>
```

- GoalCheckerSelector

It is used to select the GoalChecker that will be used by the goal_checker server. It subscribes to the goal_checker_selector topic to receive command messages with the name of the GoalChecker to be used. It is commonly used before of the FollowPathAction. The selected_goal_checker output port is passed to goal_checker_id input port of the FollowPathAction. If none is provided on the topic, the default_goal_checker is used.
Any publisher to this topic needs to be configured with some QoS defined as reliable and transient local.

- ProgressCheckerSelector

It is used to select the ProgressChecker that will be used by the progress_checker server. It subscribes to the progress_checker_selector topic to receive command messages with the name of the ProgressChecker to be used. It is commonly used before of the FollowPathAction. The selected_progess_checker output port is passed to

progress_checker_id input port of the FollowPathAction. If none is provided on the topic, the default_progress_checker is used.
Any publisher to this topic needs to be configured with some QoS defined as reliable and transient local.

```
<ProgressCheckerSelector
selected_progress_checker="{selected_progress_checker}"
default_progress_checker="precise_progress_checker"
topic_name="progress_checker_selector"/>
```

- NavigateThroughPoses

Invokes the NavigateThroughPoses ROS 2 action server, which is implemented by the bt_navigator module.

```
<NavigateThroughPoses goals="{goals}" server_name="NavigateThroughPoses"
server_timeout="10" error_code_id="{navigate_through_poses_error_code}"
behavior_tree="<some-path>/behavior_trees/navigate_through_poses_w_replanning_a
nd_recovery.xml"/>
```

- ComputePathThroughPoses

Invokes the ComputePathThroughPoses ROS 2 action server, which is implemented by the nav2_planner module. The server address can be remapped using the server_name input port.

```
<ComputePathThroughPoses goals="{goals}" path="{path}" planner_id="GridBased"
server_name="ComputePathThroughPoses" server_timeout="10"
error_code_id="{compute_path_error_code}"/>
```

- ComputeCoveragePath

Invokes the ComputeCoveragePath ROS 2 action server, which is implemented by the opennav_coverage server module. The server address can be remapped using the server_name input port. This server can take in both cartesian and GPS coordinates and is implemented using the Fields2Cover library.

```
<ComputeCoveragePath file_field="{field_filepath}" nav_path="{path}"
coverage_path="{cov_path}" server_name="ComputeCoverage" server_timeout="10"
error_code_id="{compute_coverage_error_code}"/>
```

- CancelCoverage

Used to cancel the goals given to the complete coverage action server. The server address can be remapped using the server_name input port.

```
<CancelCoverage server_name="compute_complete_coverage" server_timeout="10"/>
```

- RemovePassedGoals

Looks over the input port goals and removes any point that the robot is in close proximity to or has recently passed. This is used to cull goal points that have been passed from ComputePathToPoses to enable replanning to only the current task goals.

```
<RemovePassedGoals radius="0.6" input_goals="{goals}" output_goals="{goals}"/>
```

- CancelControl

Used to cancel the goals given to the controllers' action server. The server address can be remapped using the server_name input port.

```
<CancelControl server_name="FollowPath" server_timeout="10"/>
```

- CancelBackUp

Used to cancel the backup action that is part of the behavior server. The server address can be remapped using the server_name input port.

```
<CancelBackUp server_name="BackUp" server_timeout="10"/>
```

- CancelSpin

  Used to cancel the spin action that is part of the behavior server. The server address can be remapped using the server_name input port.

  ```
  <CancelSpin server_name="Spin" server_timeout="10"/>
  ```

- CancelWait

  Used to cancel the wait action that is part of the behavior server. The server address can be remapped using the server_name input port.

  ```
  <CancelWait server_name="Wait" server_timeout="10"/>
  ```

- CancelDriveOnHeading

  Used to cancel the drive on heading action that is part of the behavior server. The server address can be remapped using the server_name input port.

  ```
  <CancelDriveOnHeading server_name="drive_on_heading"
  server_timeout="10"/>
  ```

- CancelAssistedTeleop

  Used to cancel the AssistedTeleop action that is part of the behavior server. The server address can be remapped using the server_name input port.

  ```
  <CancelAssistedTeleop server_name="assisted_teleop" server_timeout="10"/>
  ```

- SmoothPath

  Invokes the SmoothPath action API in the smoother server to smooth a given path plan.

```
<SmoothPath unsmoothed_path="{path}" smoothed_path="{path}"
max_smoothing_duration="3.0" smoother_id="simple_smoother"
check_for_collisions="false"
smoothing_duration="{smoothing_duration_used}"
was_completed="{smoothing_completed}"
error_code_id="{smoothing_path_error_code}"/>
```