DTU

Rasmus Andersen

34761 – Robot Autonomy

Behavior Trees

Overview of 34761 – Robot Autonomy

- 3 lectures on software frameworks
- 7 lectures on building your own autonomy stack for a mobile robot
- 1 lecture on DL/RL – an overview of black-box approaches to what you have done
- 2 lectures of project work before hand in + guest lectures

Software frameworks | Robot autonomy stack | Project work + guest lectures

30th Jan	6th Feb	13th Feb	20th Feb	27th Feb	6th Mar	13th Mar	20th Mar	27th Mar	2nd Apr	9th Apr	16th Apr	23rd Apr	30th April
Software architecture	Intro to ROS2	More on ROS2	Localization	Mapping	Localization wrt. maps	SLAM	Navigation	Easter	Exploration	Behavior Trees	Black Box Approaches	Group work	

Today

Exam / Evaluation

- To successfully complete the course, you need to:
  1. Submit a group report about your final project
     - Students are expected to work in their groups
     - Implement your own version of a set of selected topics from the course material
     - Hand in an approximately 4-6-page paper of the work including a (video) demonstration
       – Use standard IEEE template format
  2. Passing the report will be a prerequisite to join the final exam questionnaire

- Important dates:
  – **Deadline for forming groups: 13th February 2024**
  – **Deadline for handing in report: 1st May at 23:55 2024**
  – **Notification of failed reports: 8th May at 16:00 2024**
  – **Examination: 22nd May 2024 (NOTE this has been updated)**

What should be in the report

1) Your objective – what is it you want to implement
   – Minimum 1 of the course topics on autonomy

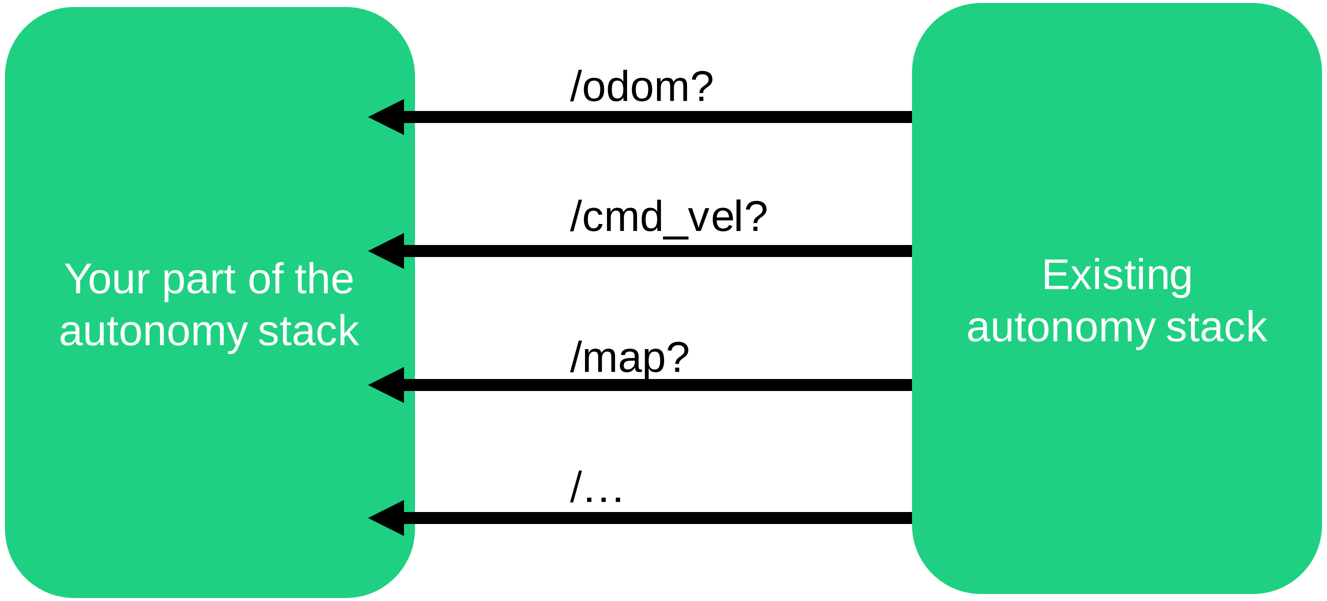
Software frameworks | Robot autonomy stack | Project work + guest lectures

30th Jan	6th Feb	13th Feb	20th Feb	27th Feb	6th Mar	13th Mar	20th Mar	27th Mar	2nd Apr	9th Apr	16th Apr	23rd Apr	30th April
Software architecture	Intro to ROS2	More on ROS2	Localization	Mapping	Localization wrt. maps	SLAM	Navigation	Easter	Exploration	Behavior Trees	Black Box Approaches	Group work	

What should be in the report

1) Your objective – what is it you want to implement
   – Minimum 1 of the course topics on autonomy

2) Your prerequisites – what assumptions have you made
   – You can use the existing autonomy stack for the parts you don't plan on implementing (e.g. if you want to implement mapping, you can use the odometry already provided)



What should be in the report

Accuracy is secondary to your method!

1) Your objective – what is it you want to implement
   – Minimum 1 of the course topics on autonomy

2) Your prerequisites – what assumptions have you made
   – You can use the existing autonomy stack for the parts you don't plan on implementing (e.g. if you want to implement mapping, you can use the odometry already provided)

3) Your approach – how do you plan on implementing it
   – Background theory of the approach you want to implement

4) Your implementation – how did you actually implement it (ROS nodes, topics, timers etc.)

5) Your results – a description of the behavior of the robot + a video of your robot driving in the simulation

Groups

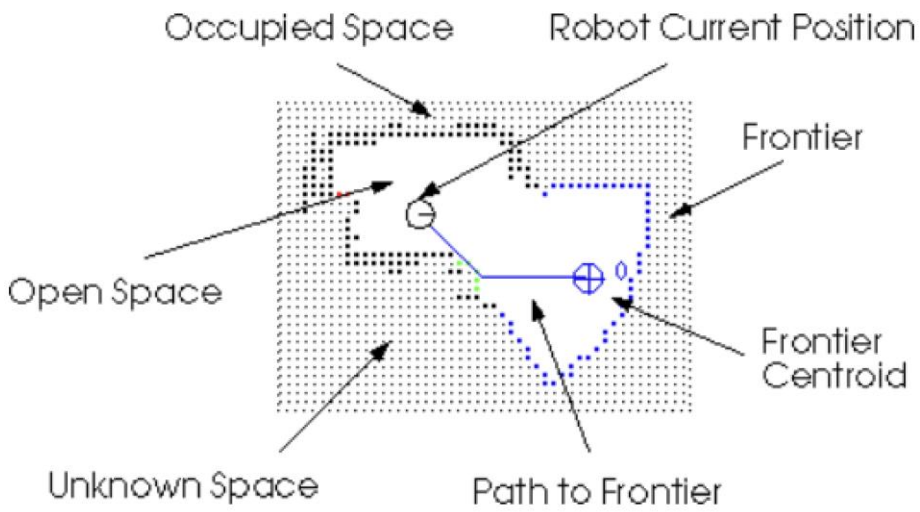
- Groups are created on LEARN, where you also hand in the report
  – Please check all your group members are there

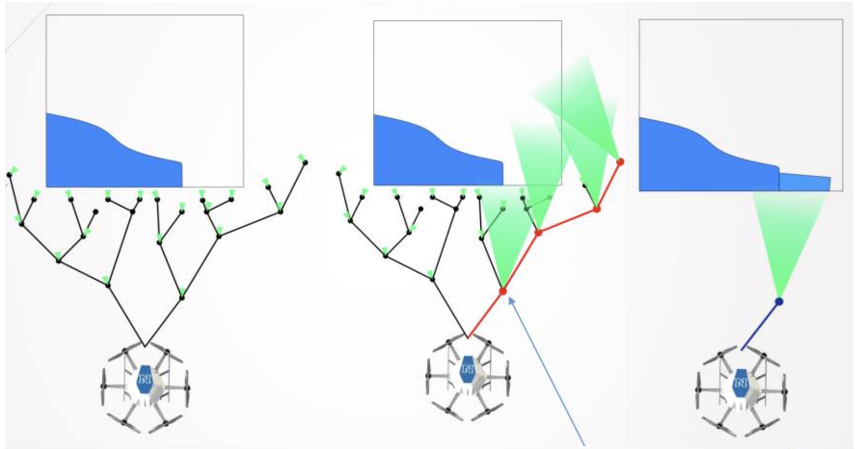
- For those who haven’t signed up for a group, do so ASAP!
  – And send me your student # and which group you have joined

  – I will group together those still missing a group on Friday

Recall from last lecture

- Frontier exploration
- Receding horizon next-best view planner



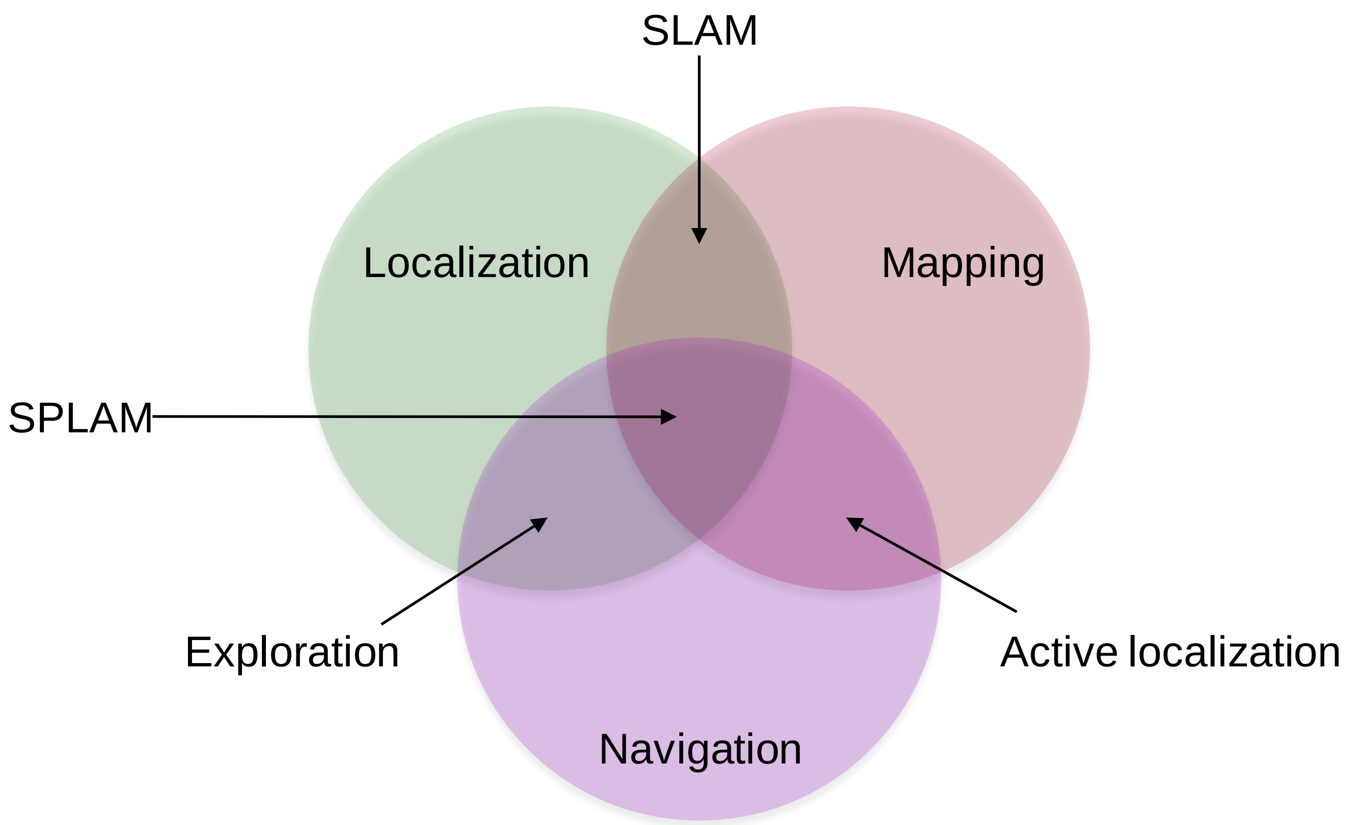


$$\mathbf{Gain}(n_k) = \mathbf{Gain}(n_{k-1}) + \mathbf{Visible}(\mathcal{M}, \xi_k) e^{-\lambda c(\sigma_{k-1}^k)}$$

Outline for the next 7 weeks

- Our own autonomy stack:
  1. Localization
  2. Mapping
  3. Navigation
  4. Exploration
  5. Behaviour trees

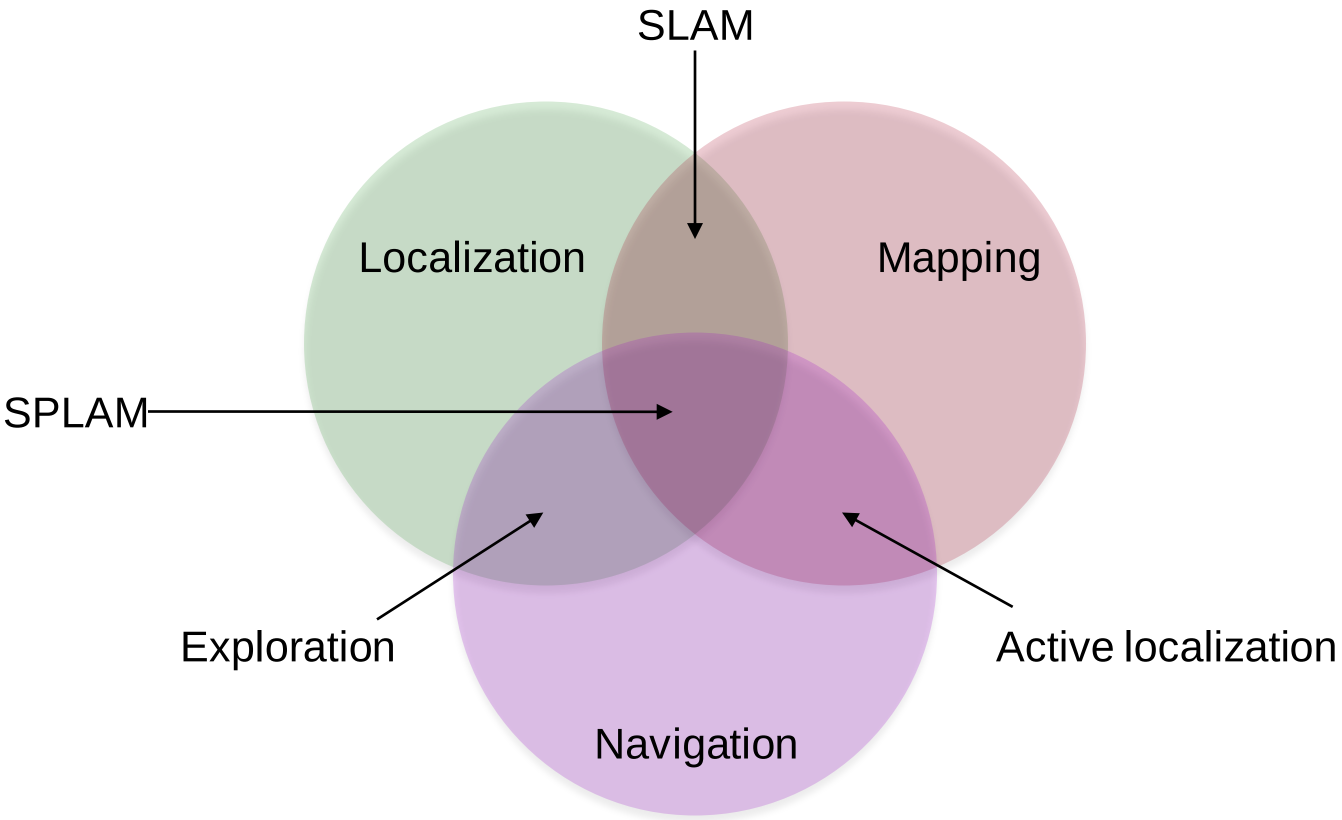
Topic of today



Outline for the next 7 weeks

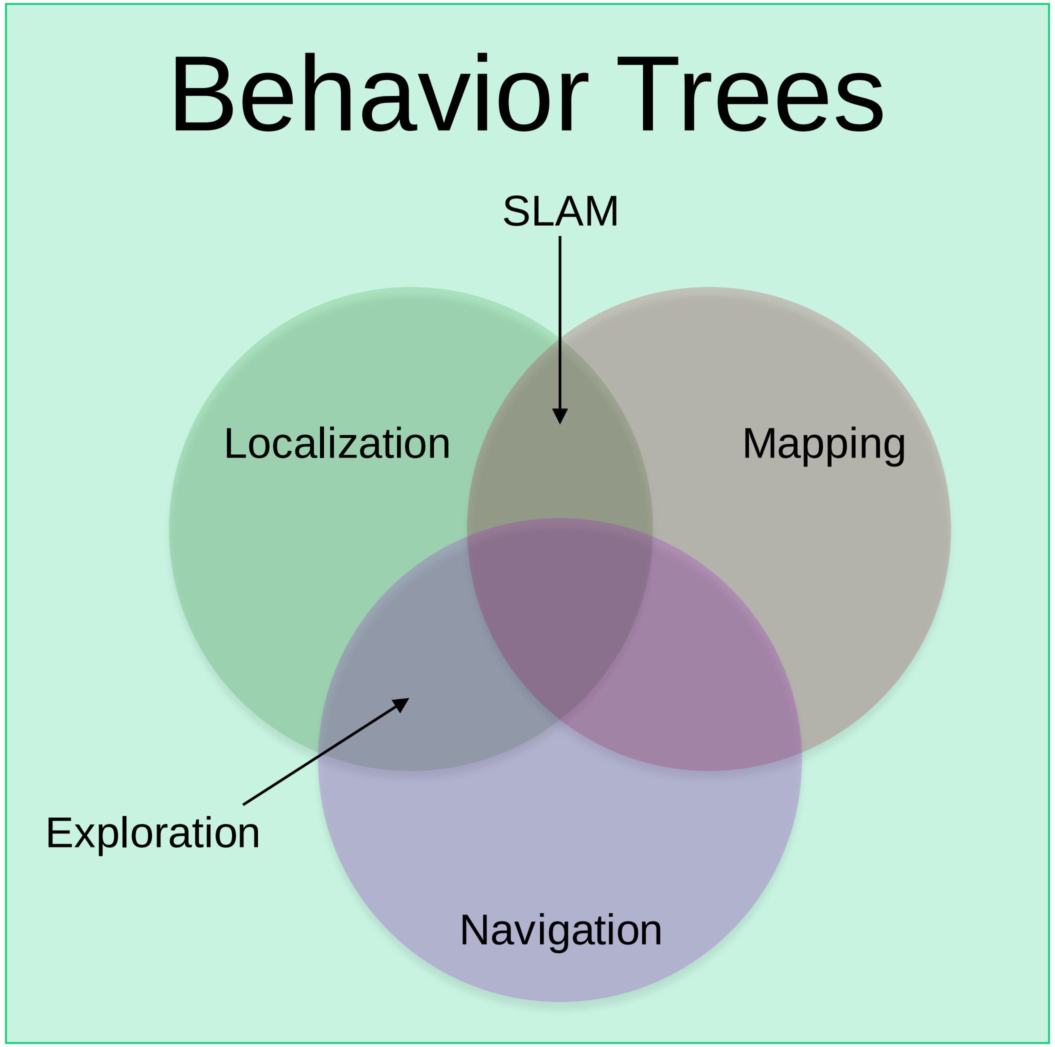
- Our own autonomy stack:
  1. Localization
  2. Mapping
  3. Navigation
  4. Exploration
  5. Behaviour trees
  – Rule-based systems
  – Hierarchical Task Networks
  – Skill-based Systems
  – Behavior Trees

Topic of today

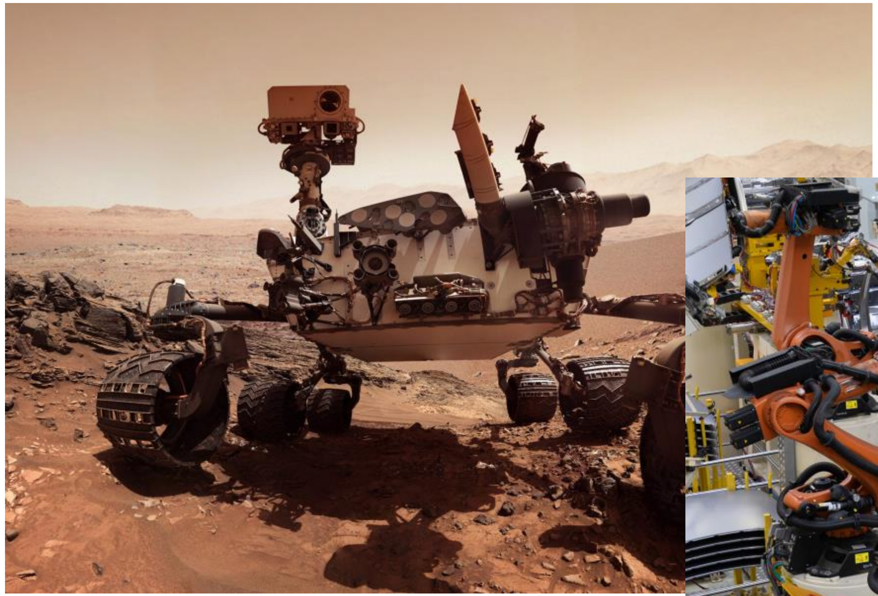


We now have the basic components for a robot

Now what?



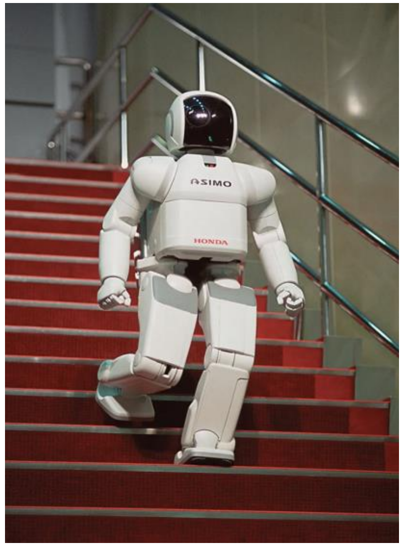
Operation of Autonomous systems?

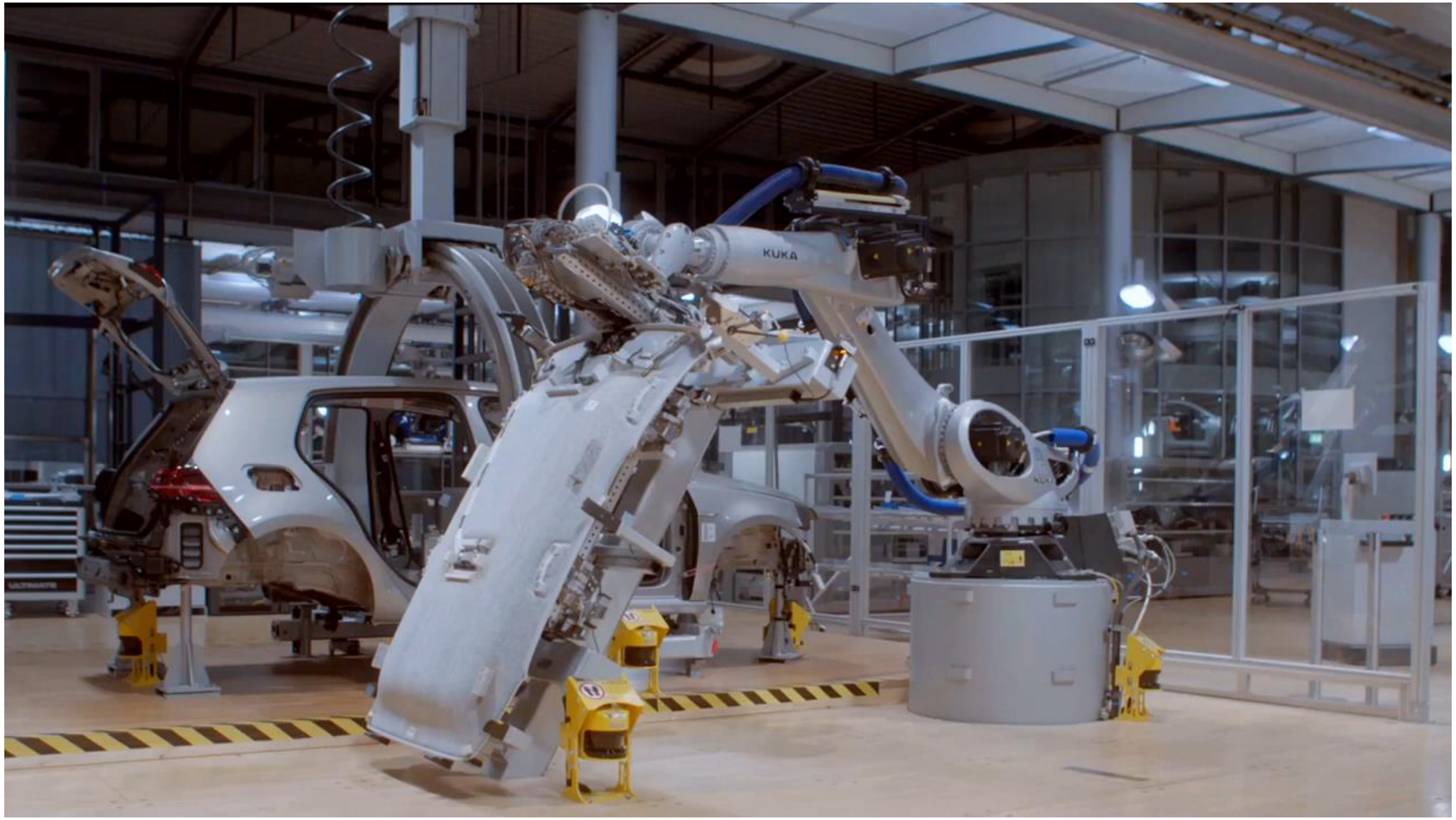


Uncertain world?

A high-level decision-making system

- Structure when to do what
- Deal with shifting objectives
- Handle unforeseen conditions



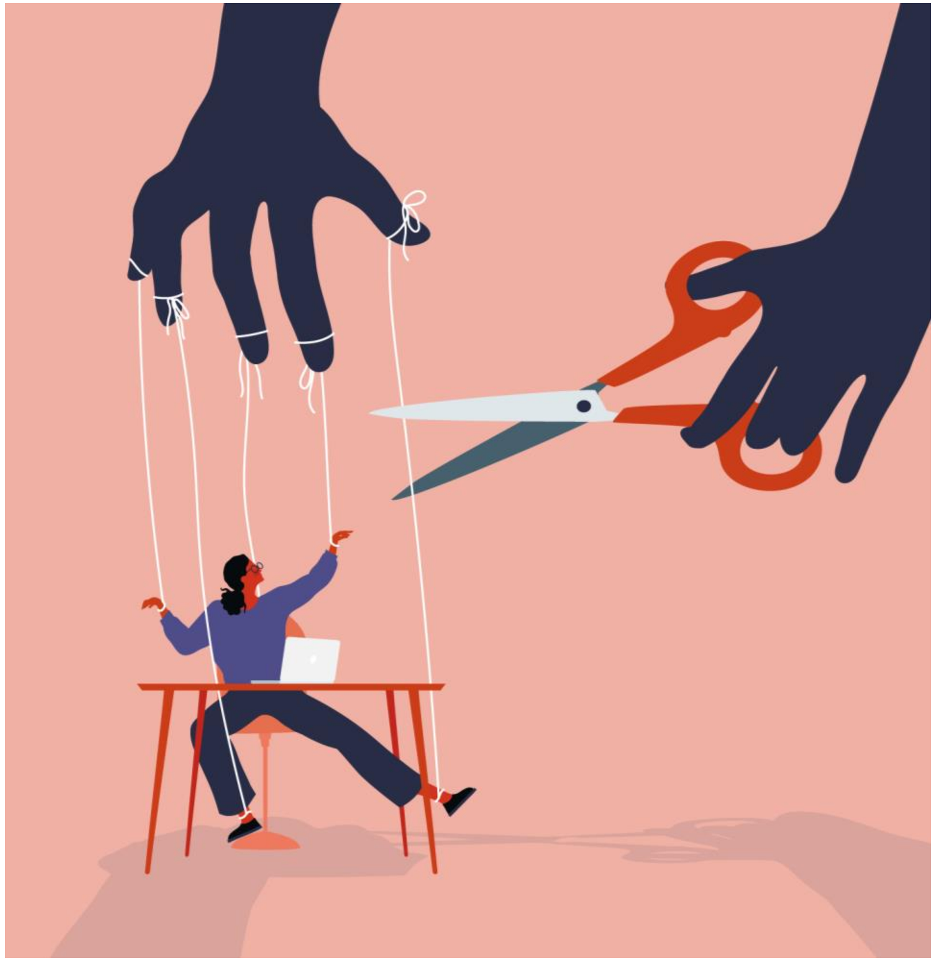

KUKA

MiR

MOBILE INDUSTRIAL ROBOTS

How do you structure autonomy?

- Rule-based systems
- Hierarchical Task Networks
- Skill-based Systems
- Behavior Trees



Rule-based systems

Rule-based systems

- IF THEN setup
  - Pair situations/states (the IF) with actions (the THEN)

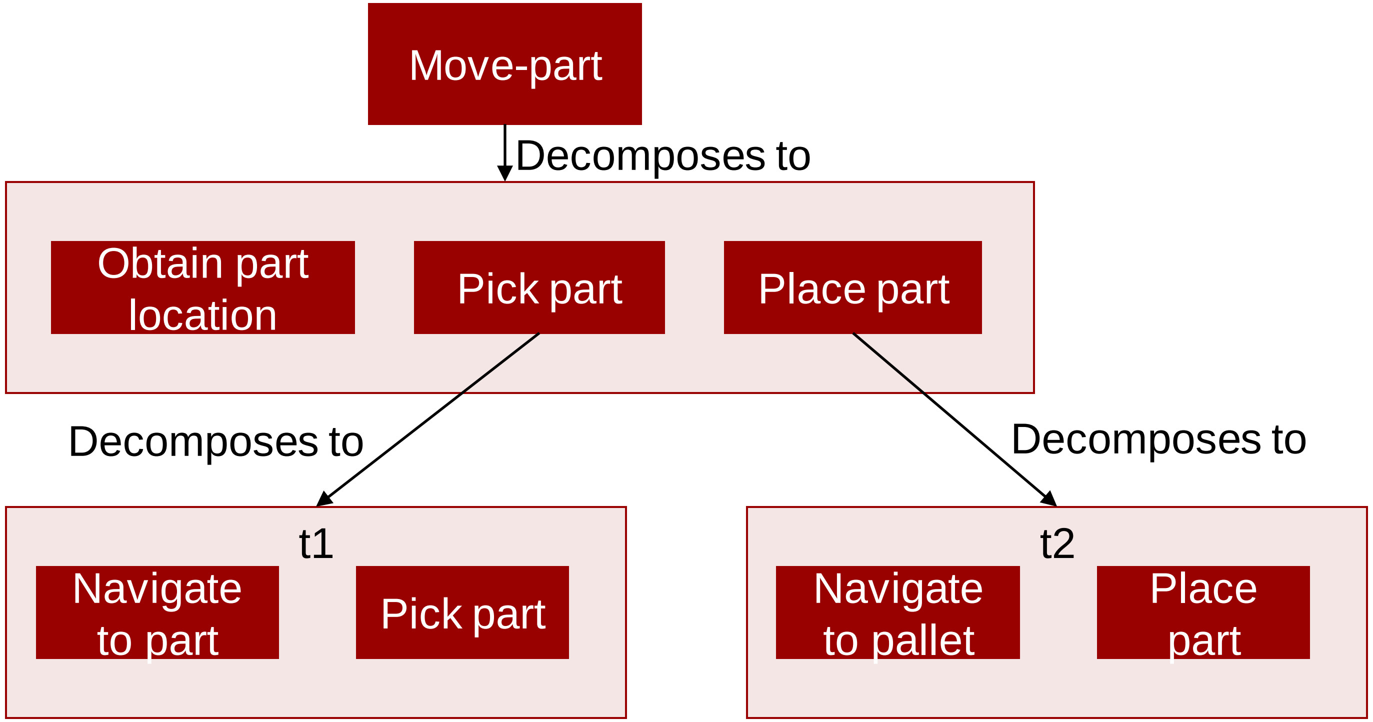
- Very often used in simpler/smaller “AI” systems (tic-tac-toe, etc.)

- Heavily expert knowledge driven (i.e., fact-driven)
  - Relies on expert knowledge to define suitable rules
  - The problem can be too hard to handcraft rules for
  - The search space increases exponentially (curse of dimensionality)

Hierarchical Task Networks

- Create a plan consisting of tasks
- Break high level tasks down into smaller tasks recursively
- Continue until you reach simple action-primitives
- An HTN method is a 4-tuple *m=(name(m), task(m), subtasks(m), constraints(m))*
  - *name(m)*
    - The name of the method
  - *task(m)*
    - A non-primitive task
  - *subtasks(m)*
    - Subtasks to break down the method
  - *constraints(m)*
    - Can be any kind of constraints, e.g. time, resources, preconditions, etc.

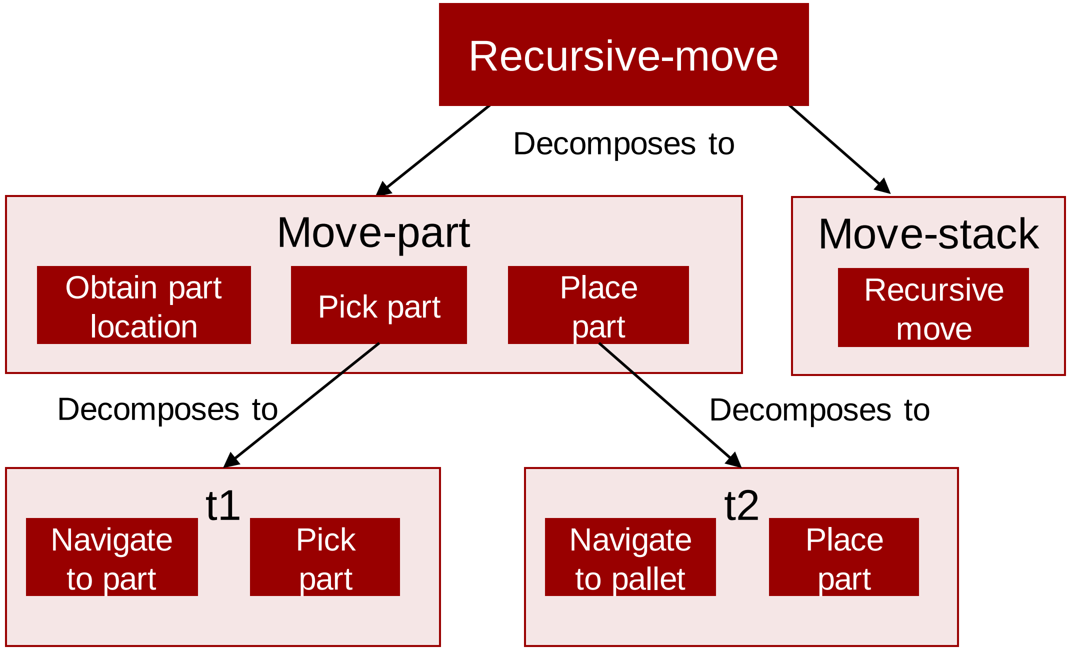
Hierarchical Task Networks – Pick-and-place



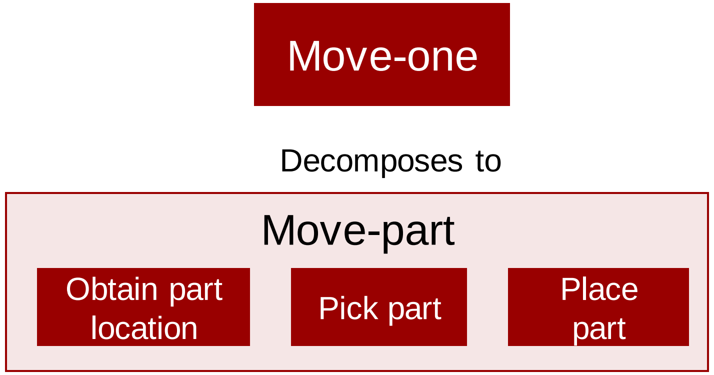
- Constraints(stack pallet): (t1 > t2)

Hierarchical Task Networks – Pick-and-place

- Move stack: repeatedly move part onto pallet



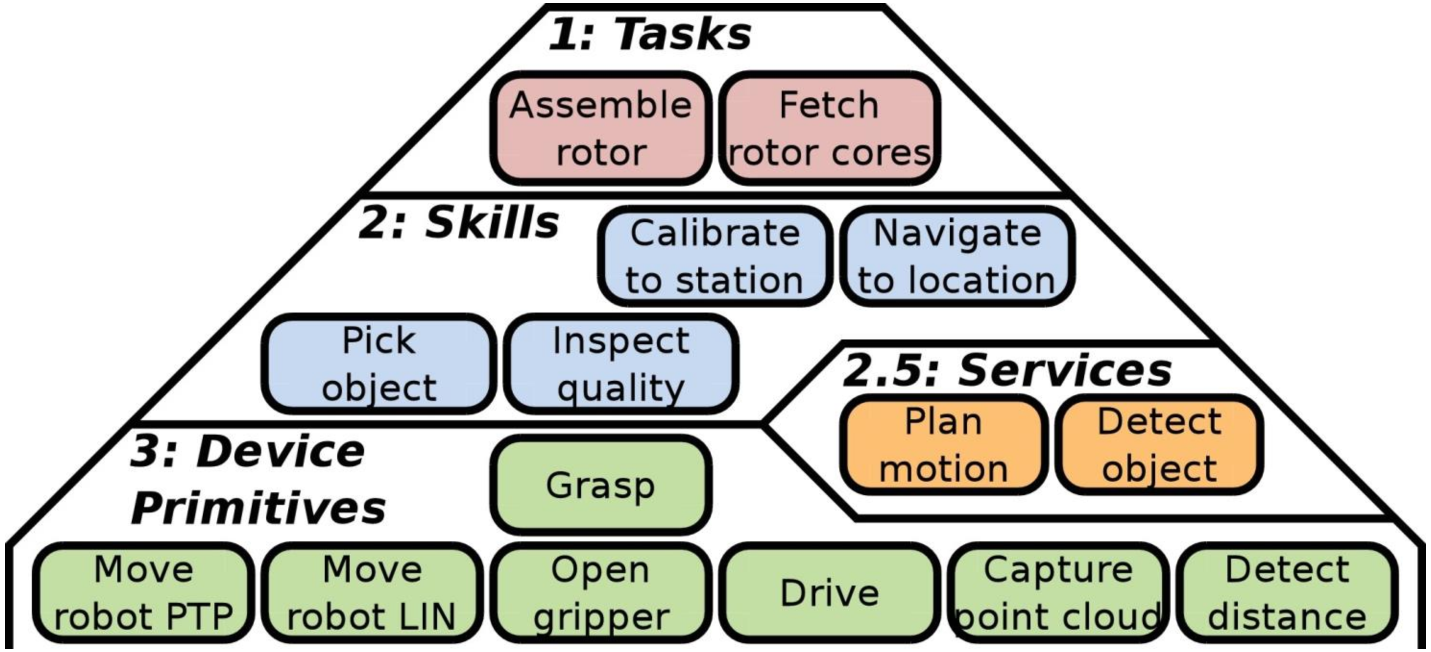
Constraints(move-stack): {before(t1, number of parts left>1), before(t2, pallet has space)}



Constraints(move-one): {before(t1, pallet stack=1), before(t2, pallet has space)}

Skill-based approaches

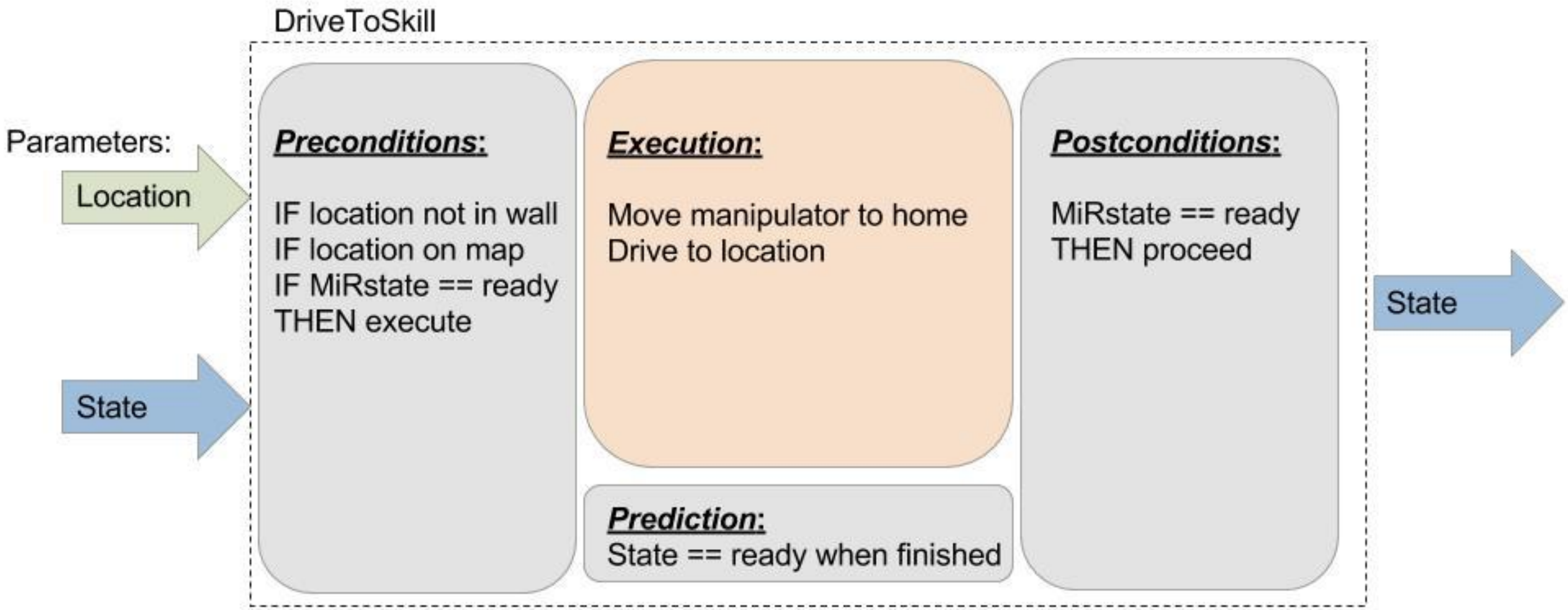
- Abstract low-level device/motion-primitives into sets that compose a more intuitive action
- Typically, most interesting for collaborative robot systems (e.g., industrial robots)



Schou, C., Andersen, R. S., Chrysostomou, D., Bøgh, S., & Madsen, O. (2018). Skill-based instruction of collaborative robots in industrial settings. *Robotics and Computer-Integrated Manufacturing*, *53*, 72-80.

Skill-based approaches

- Parameterizable and task-related actions of the robot
- Task programming is conducted by sequencing skills
- Composed of precondition checks, execution, and postcondition checks



Behavior Trees – the basics

- A decision tree
  - Execution order is determined by the leaf composition (often left → right)
  - Structures how the system/agent should switch between behaviors
  - Recursively updates the tree at specified frequencies (ticks)

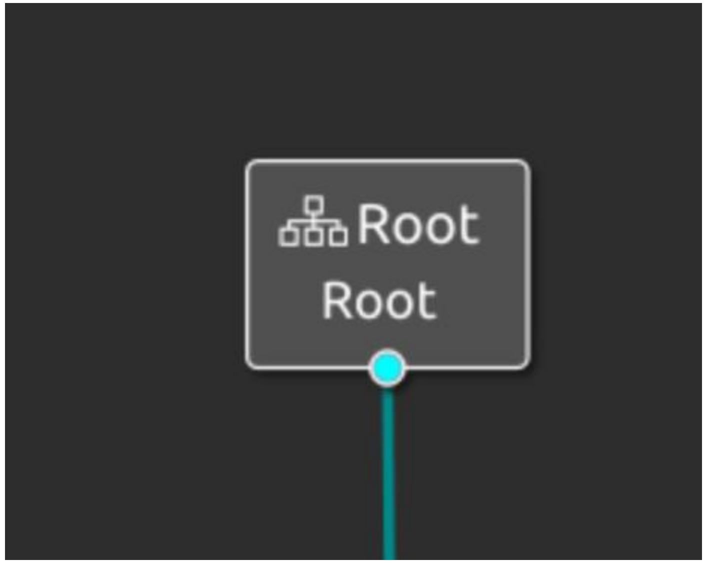
- Contains mainly three types of nodes;
  - **The root node**
  - **Control flow nodes**
  - **Execution nodes**

- A node returns either *success, failure,* or, *running*

- *Note: the notation vary across the literature, but the overall philosophy remains the same*

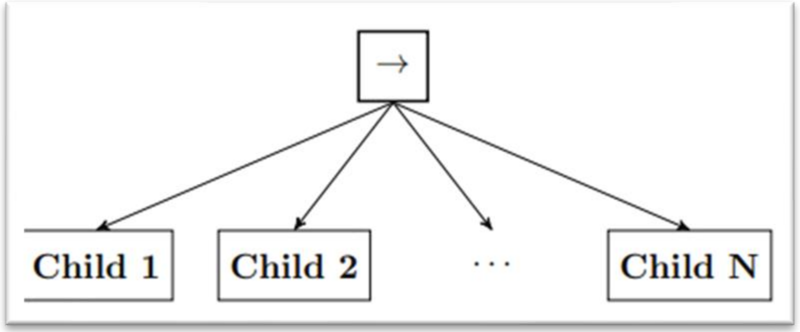
Behavior Tree – Root node

- **The root node**
  - Initial state of the program
  - Has exactly one child in the tree

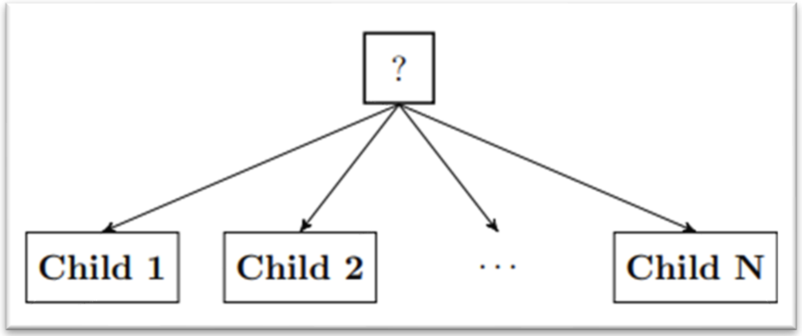


Behavior Trees – Control flow nodes

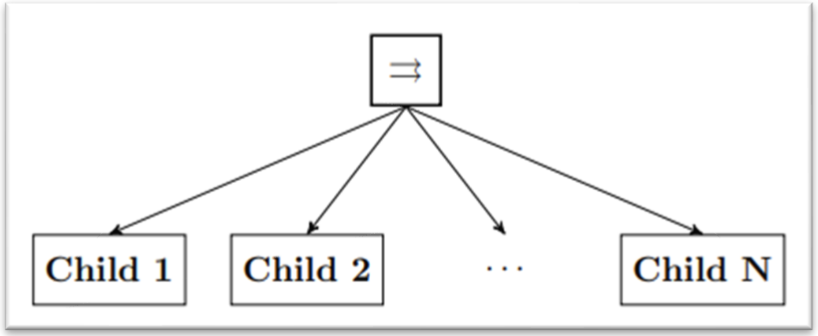
- Sequence
  - Route the ticks from left to right until a child returns *running* or *failure* (halts execution)
  - Return *success* only if all children returned *success*
- Fallback
  - Route the ticks from left to right until a child returns *success* or *running* (halts execution)
  - Return *failure* only if all children returned *failure*
- Parallel
  - Route the ticks to all children
  - Return *success* only if M children returns *success,* otherwise return *failure*



Sequence node



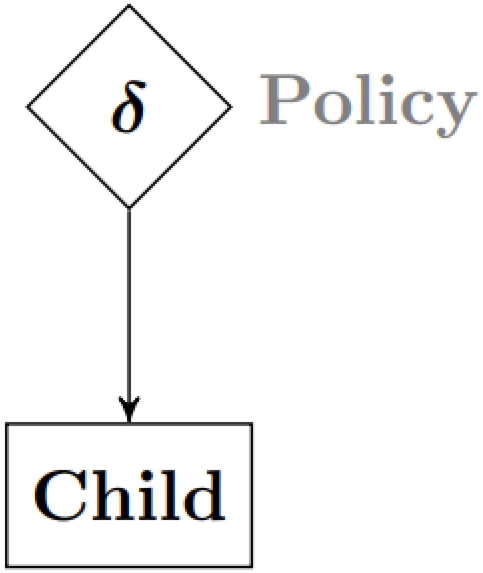
Fallback node



Parallel node

Behavior Trees – Execution nodes

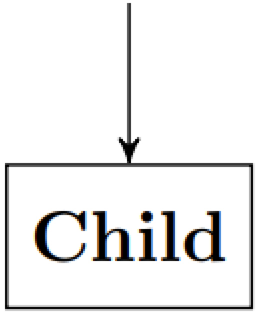
- Action
  - Execute some action/task/command
  - Usually more complex operations (pick-up object, navigate to position, etc.)
  - Returns *success, failure,* or *running* at each tick it receives
- Condition
  - Simple checks (get battery level, state of gripper, etc.)
  - Returns *success* or *failure*
- Decorator
  - Alters the return value of a child according to some policy
  - The return value is custom depending on the policy



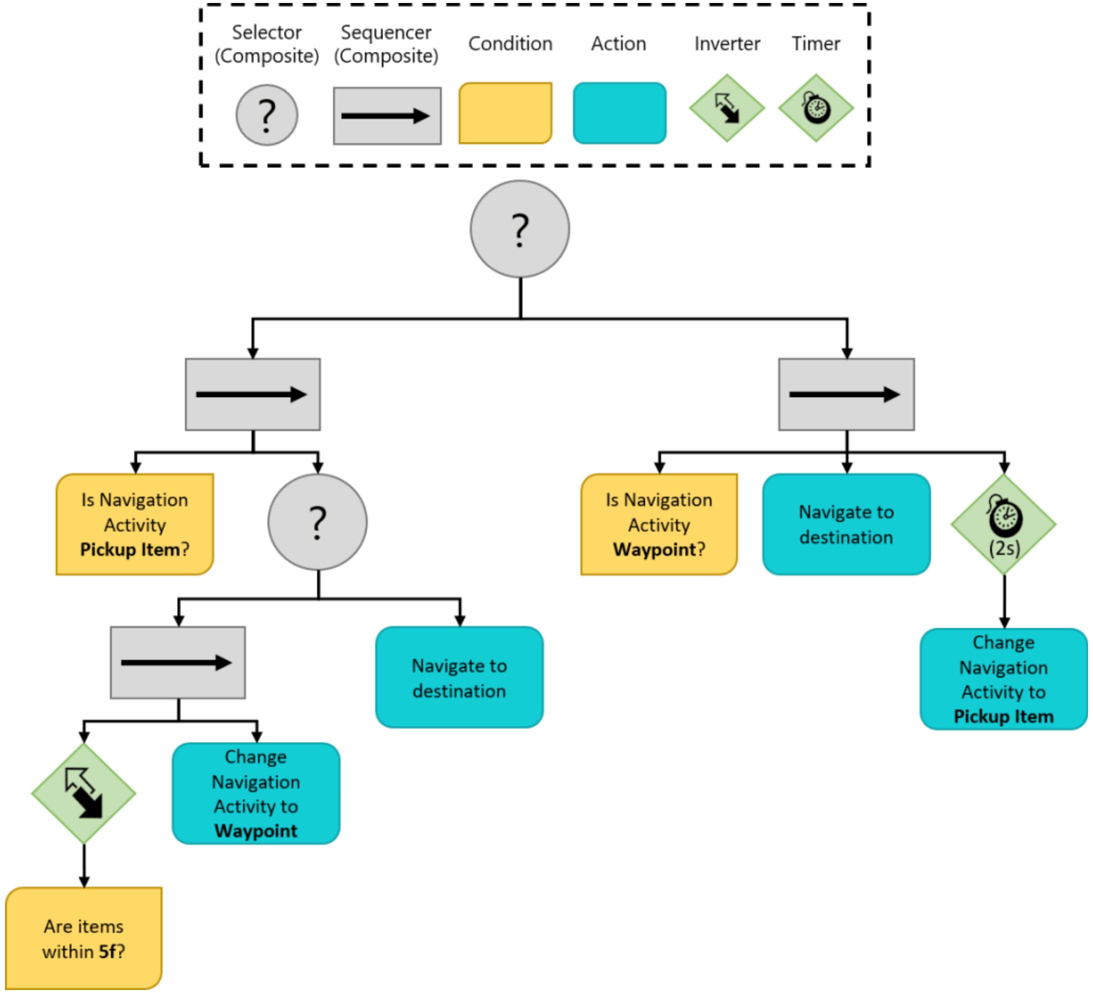
Behavior Trees – Execution nodes

- Action

Node type	Symbol	Succeeds	Fails	Running
Fallback	$\boxed{?}$	If one child succeeds	If all children fail	If one child returns Running
Sequence	$\boxed{\rightarrow}$	If all children succeed	If one child fails	If one child returns Running
Parallel	$\boxed{\rightrightarrows}$	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	$\boxed{\text{text}}$	Upon completion	If impossible to complete	During completion
Condition	(text)	If true	If false	Never
Decorator	$\Diamond$	Custom	Custom	Custom



Behavior Trees



- Gives us a structured way of representing autonomy
- Easy to implement with a well-defined control flow
- Requires definition of action and condition blocks

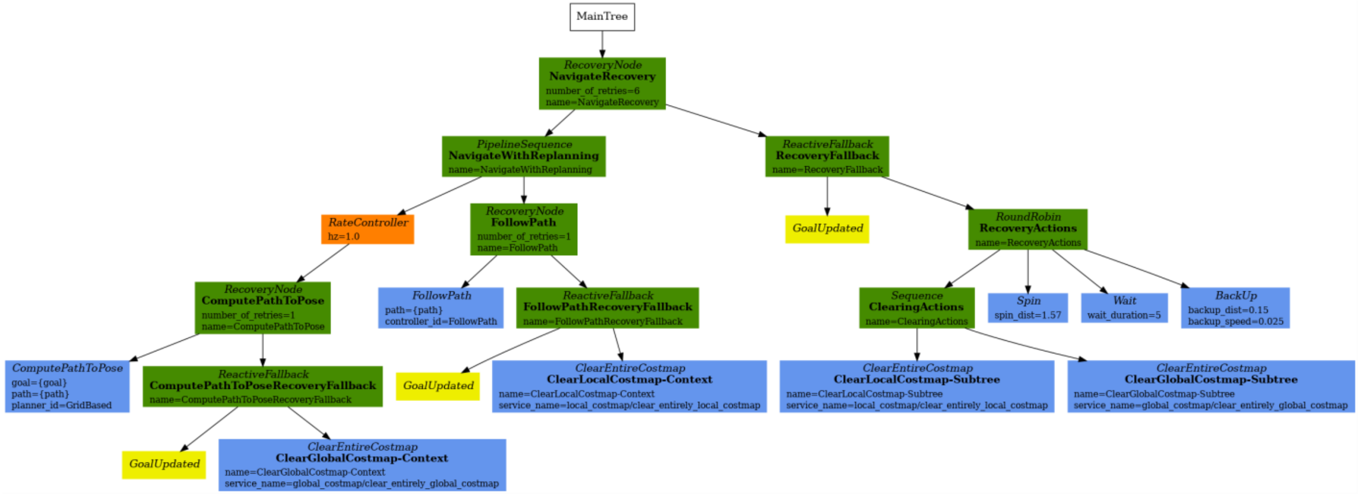
Node type	Symbol	Succeeds	Fails	Running
Fallback	$\boxed{?}$	If one child succeeds	If all children fail	If one child returns Running
Sequence	$\boxed{\rightarrow}$	If all children succeed	If one child fails	If one child returns Running
Parallel	$\boxed{\rightrightarrows}$	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	$\Diamond$	Custom	Custom	Custom

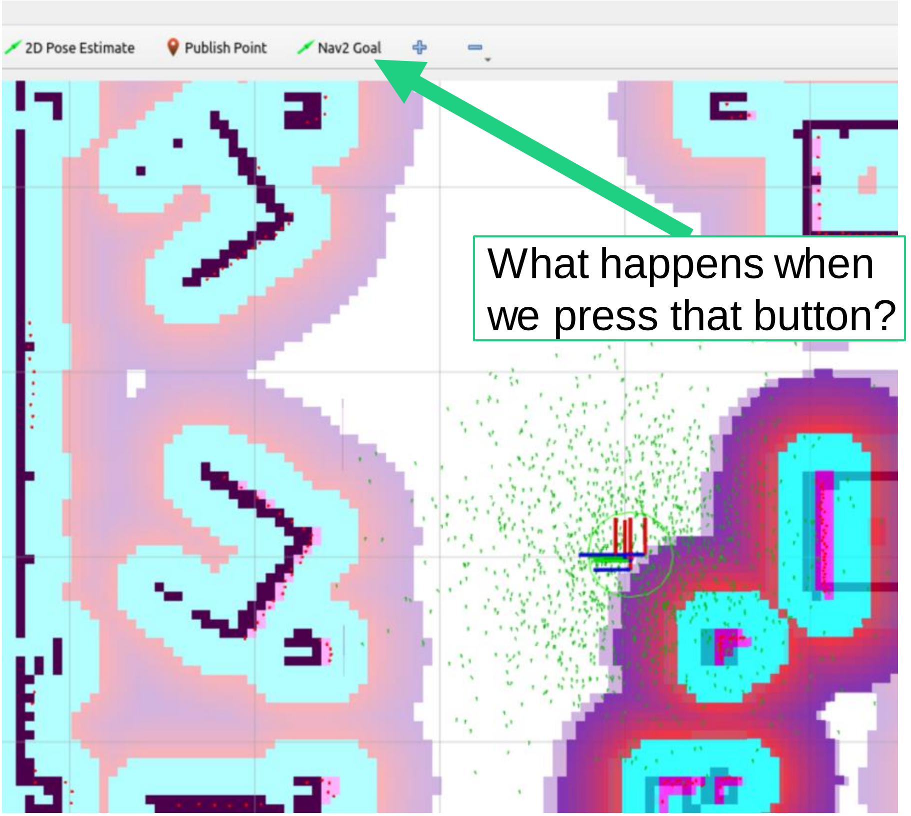
Behavior trees in ROS2

- The navigation stack already comes with a list of predefined behaviors we could use

- The default one is *navigate_to_pose_w_replanning_and_recovery.xml*

- Behavior trees are generally defined in .xml format, which is also what ROS uses



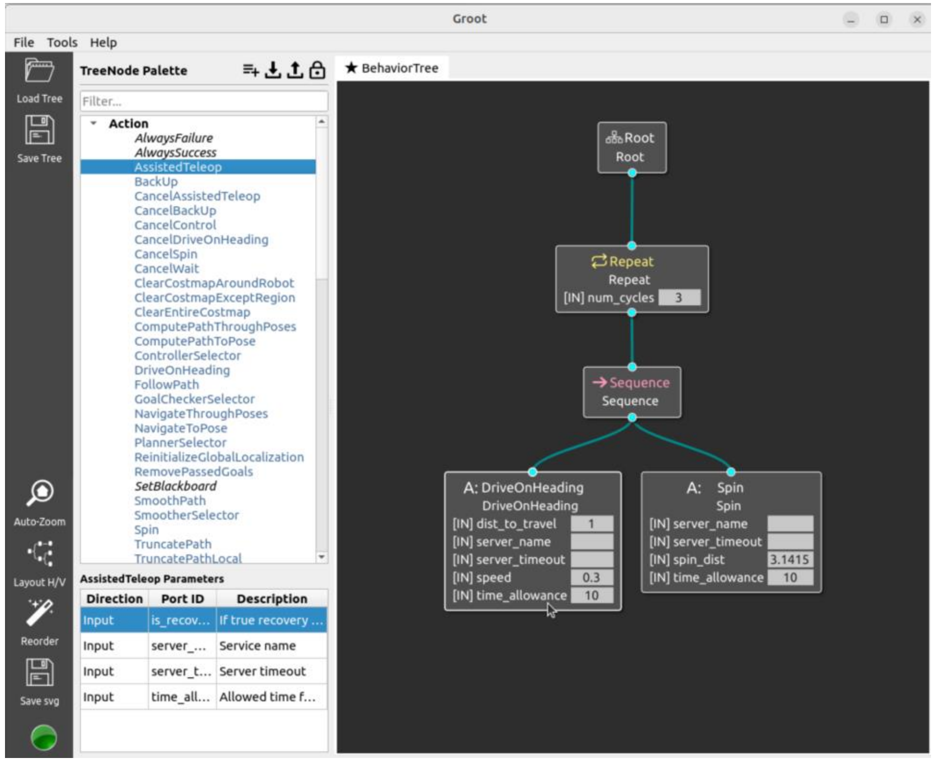


What happens when we press that button?

- follow_point.xml
- nav_to_pose_with_consistent_replanning_and_if_path_becomes_invalid.xml
- navigate_through_poses_w_replanning_and_recovery.xml
- navigate_to_pose_w_replanning_and_recovery.xml
- navigate_to_pose_w_replanning_goal_patience_and_recovery.xml
- navigate_w_recovery_and_replanning_only_if_path_becomes_invalid.xml
- navigate_w_replanning_distance.xml
- navigate_w_replanning_only_if_goal_is_updated.xml
- navigate_w_replanning_only_if_path_becomes_invalid.xml
- navigate_w_replanning_speed.xml
- navigate_w_replanning_time.xml
- odometry_calibration.xml

Groot

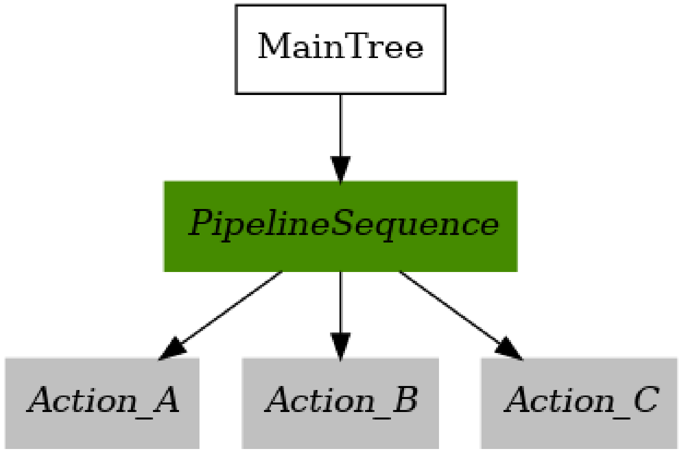
- To create a tree visually, we can use Groot
- The ‘vocabulary’ is bigger than for standard behavior trees we saw before, but the principles are the same
- Since Groot is just a general-purpose BT generation tool, we have to also specify the actions that we can use
  - This is already done for us, but if we wanted to create something custom, we would need to do this



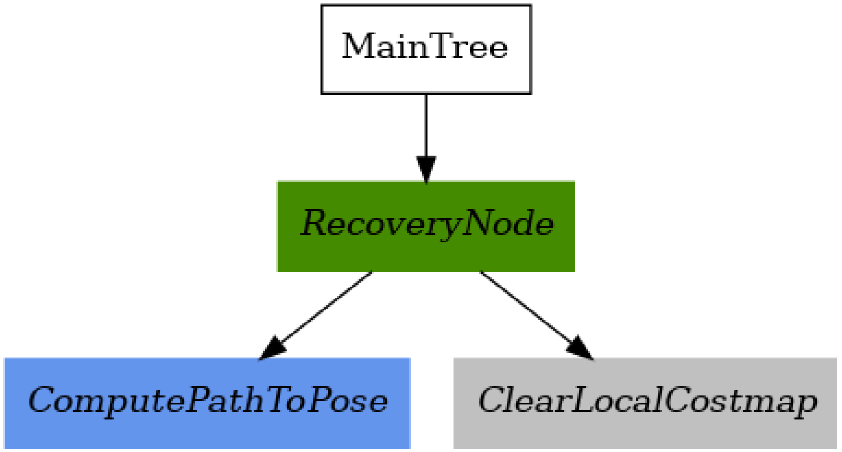

```xml
<root>
  <TreeNodesModel>
    <!-- ################################ ACTION NODES ################################# -->
    <Action ID="BackUp">
      <input_port name="backup_dist">Distance to backup</input_port>
      <input_port name="backup_speed">Speed at which to backup</input_port>
      <input_port name="time_allowance">Allowed time for reversing</input_port>
      <input_port name="server_name">Server name</input_port>
      <input_port name="server_timeout">Server timeout</input_port>
    </Action>

    <Action ID="DriveOnHeading">
      <input_port name="dist_to_travel">Distance to travel</input_port>
      <input_port name="speed">Speed at which to travel</input_port>
      <input_port name="time_allowance">Allowed time for reversing</input_port>
      <input_port name="server_name">Server name</input_port>
      <input_port name="server_timeout">Server timeout</input_port>
    </Action>
```

...

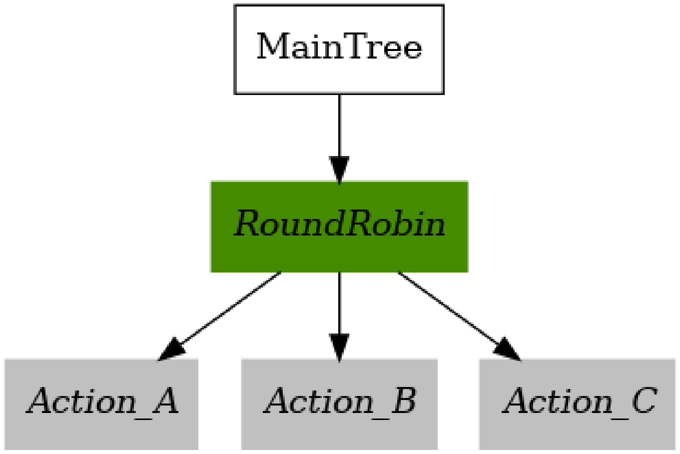
The more advanced features of ROS BTs



- Re-ticks previous children when a child returns RUNNING
- If at any point a child returns FAILURE, all children will be halted and the parent node will also return FAILURE
- Upon SUCCESS of the last node in the sequence, this node will halt and return SUCCESS



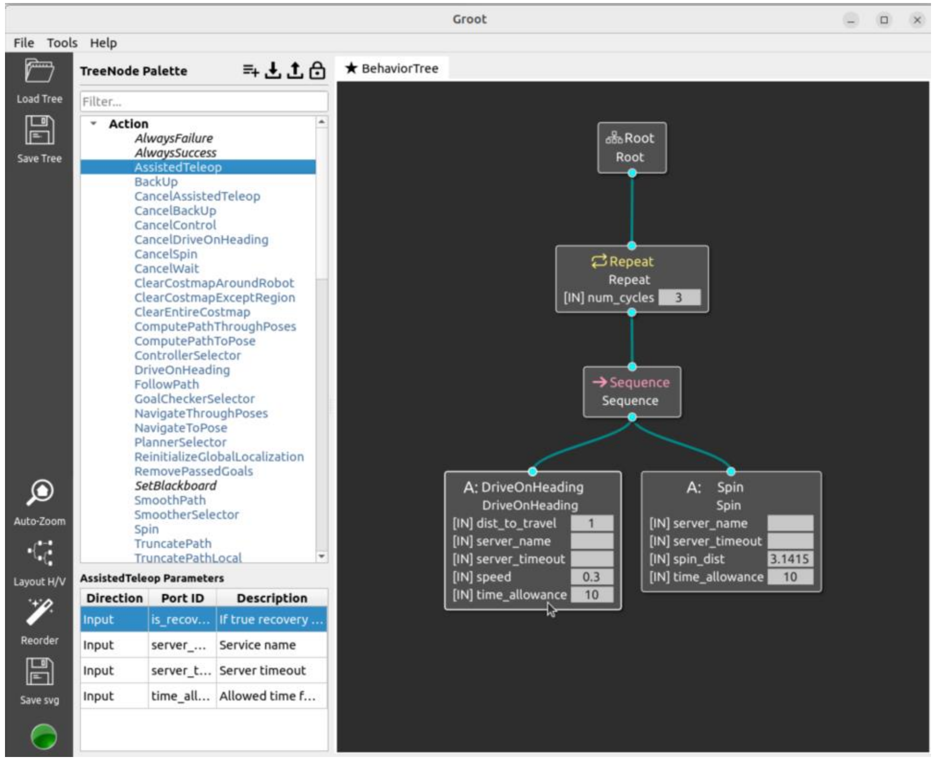
- Has only two children and returns SUCCESS if and only if the first child returns SUCCESS
- If the first child returns FAILURE, the second child will be ticked. This loop will continue until either:
  - The first child returns SUCCESS (which results in SUCCESS of the parent node)
  - The second child returns FAILURE (which results in FAILURE of the parent node)
  - The number_of_retries input parameter is violated



- Ticks its children in a round robin fashion until a child returns SUCCESS
- If all children return FAILURE so will the parent RoundRobin

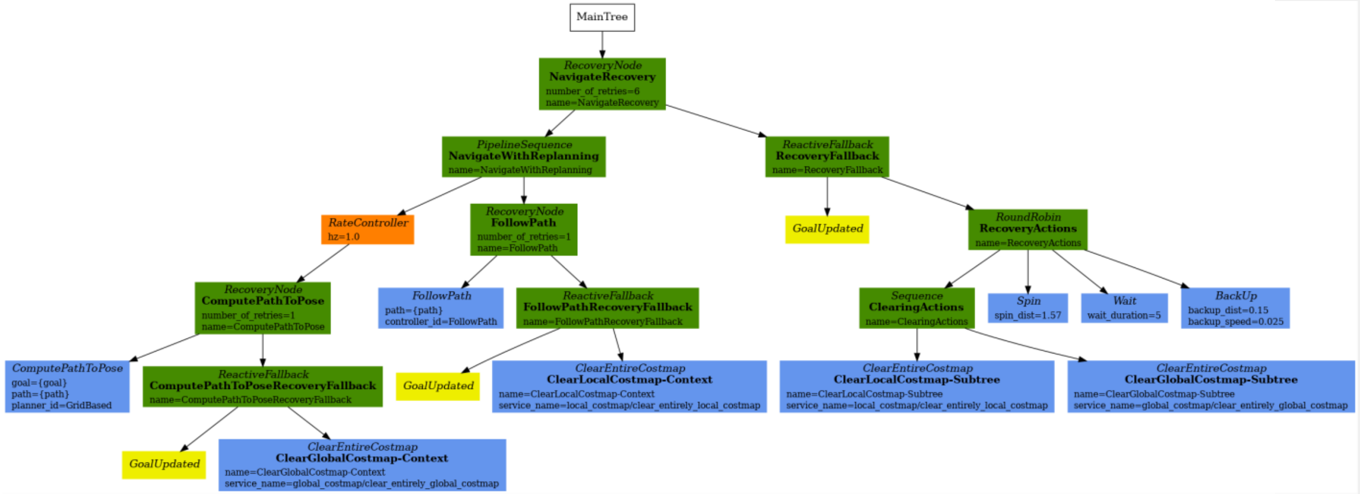
A BT from Groot

- The BT file generated with Groot will contain two parts
  - The BT itself (shown below)
  - All the node definitions (these are only used by Groot and can be ignored)



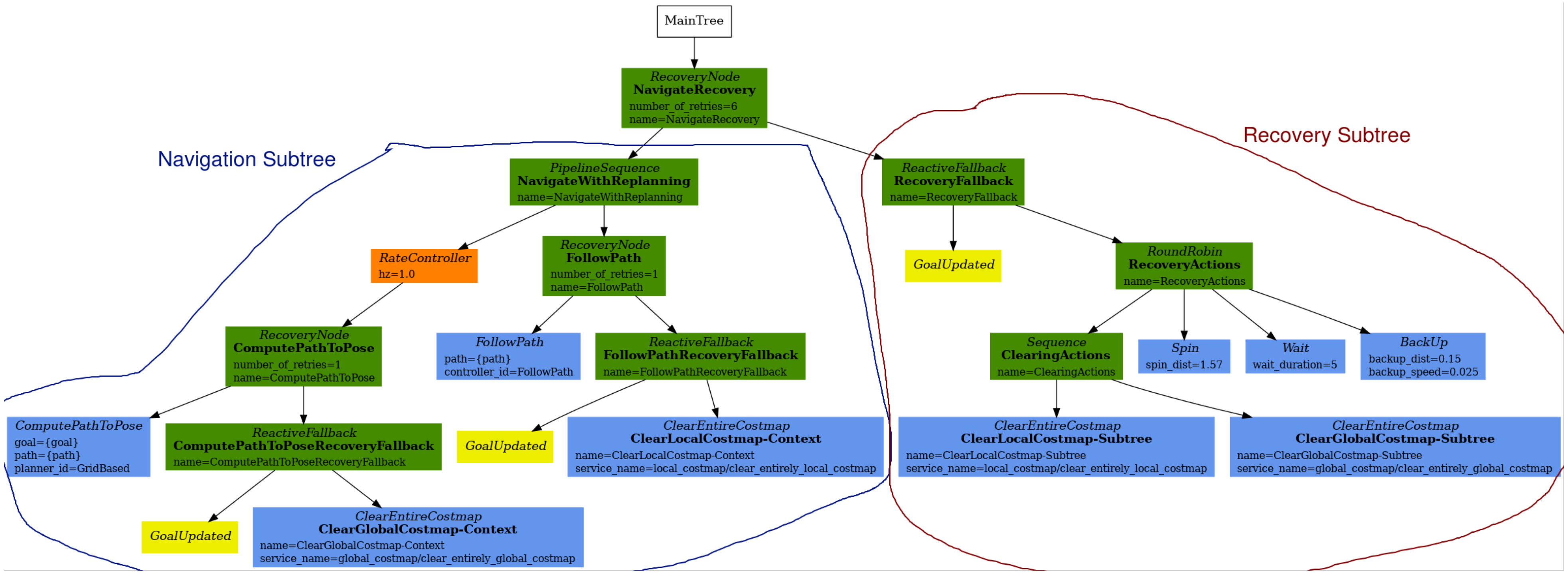

```xml
<?xml version="1.0"?>
<root main_tree_to_execute="BehaviorTree">
    <!-- ////////// -->
    <BehaviorTree ID="BehaviorTree">
        <Repeat num_cycles="3">
            <Sequence>
                <Action ID="DriveOnHeading" dist_to_travel="1" server_name="" server_timeout="" speed="0.3" time_allowance="10"/>
                <Action ID="Spin" spin_dist="3.1415" time_allowance="10"/>
            </Sequence>
        </Repeat>
    </BehaviorTree>
</root>
```

The default BT for navigation



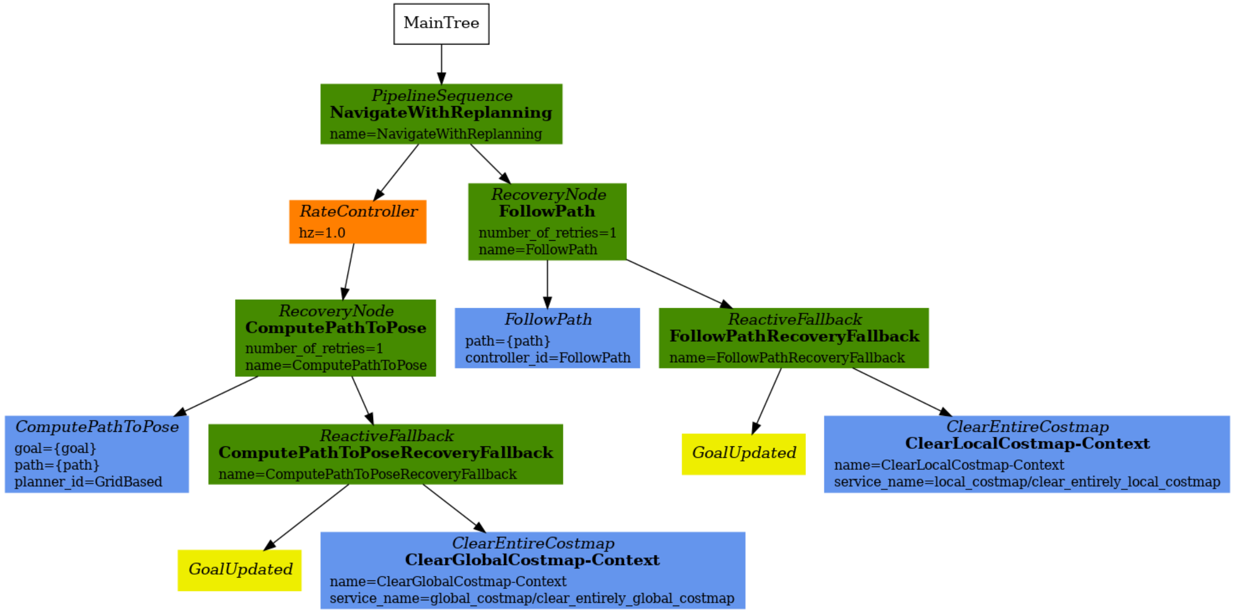

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
              <GoalUpdated/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name="global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
          </RecoveryNode>
        </RateController>
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_costmap/clear_entirely_local_costmap"/>
          </ReactiveFallback>
        </RecoveryNode>
      </PipelineSequence>
      <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="0.15" backup_speed="0.025"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

The default BT for navigation



Navigation branch

- This subtree has two primary actions *ComputePathToPose* and *FollowPath*
- If either of these two actions fail, they will attempt to clear the failure contextually
- Both the *ComputePathToPose* and the *FollowPath* follow the same general structure
  - Do the action
  - If the action fails, try to see if we can contextually recover

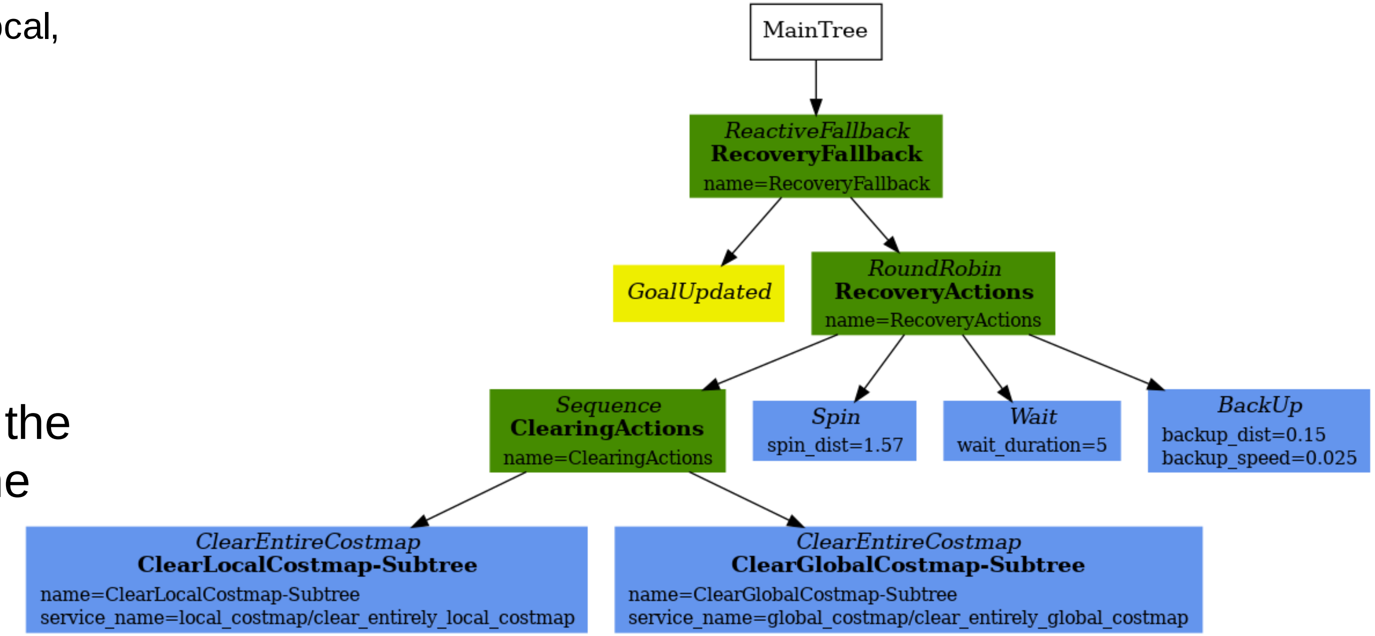


Recovery branch

- The default four system-level recoveries in the BT are:
    1. A sequence that clears both costmaps (local, and global)
    2. Spin action
    3. Wait action
    4. BackUp action

- Upon SUCCESS of any of the four children of the parent RoundRobin, the robot will attempt to renavigate in the Navigation subtree

- If this renavigation was not successful, the next child of the RoundRobin will be ticked



Behavior Trees – Example

Exercises

1. (optional) This is the exercise on behavior trees from *Introduction to Autonomous Systems* and is less complicated to set up, but gives the same functionality to a pacman environment instead of your turtlebot
   – Implement a behavior tree for a Pacman environment
      • https://github.com/RasmusAndersen/pacman

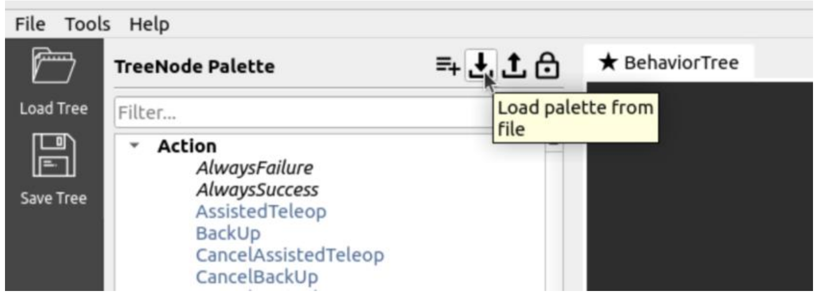
1. Create a behavior tree that makes your turtlebot drive in a square
   – Installation instructions are on the following slides
   – You should be able to solve the exercise using only 4 types of nodes:
      1. A **Repeat** decorator
      2. A **Sequence** control
      3. A **DriveOnHeading** action
      4. A **Spin** action

Installing and running Groot

- Building Groot
  - cd ~/ros2_ws/src # or where you have your workspace located
  - git clone https://github.com/BehaviorTree/Groot.git
  - cd ..
  - rosdep install --from-paths src --ignore-src
  - colcon build --symlink-install --packages-select groot
  - (source your setup.bash)

- Running Groot
  - ros2 run groot Groot

Creating a tree



1. Open Groot in editor mode
2. Select the *Load palette from file* option either via the context menu or the import icon in the top middle of the menu bar.
3. Open the file */opt/ros/humble/share/nav2_behavior_tree/nav2_tree_nodes.xml* to import all the custom behavior tree nodes used for navigation. This is the palette of Nav2 custom behavior tree nodes.
4. You can now create your own tree and save the corresponding xml file
   1. *server_timeout* and *server_name* does not have to be filled out as we have to delete them later anyway

Unfortunately, the xml file generated from Groot contains parameters that breaks the ROS navigation stack

1. Open the xml file of the tree you created
2. Delete the *server_timeout* and *server_name* parameters for both DriveOnHeading and Spin

```
<Action ID="DriveOnHeading" dist_to_travel="1" server_name="" server_timeout="" speed="0.3" time_allowance="10"/>
```

Loading a tree

To load a tree

1. Select *Load tree* option near the top left corner
2. Browse the tree you want to visualize, then select *OK*.
   – Predefined Nav2 BTs exist in */opt/ros/humble/share/nav2_bt_navigator/behavior_trees/*

Using a tree

- Download and extract params.zip to your my_turtlebot package
- Replace the path of *default_nav_to_pose_bt_xml* on line 54 in nav2_params.yaml to the path of your behaviour tree previously created
- Launch your simulation like you normally would, but with the added parameter *params_file:=/path/to/nav2_params.yaml*
  - E.g.:
    ros2 launch my_turtlebot turtlebot_simulation.launch.py params_file:=/home/ubuntu/Documents/ros2_ws/src/RobotAutonomy/params/nav2_params.yaml
- Give an initial pose estimate and press a random navigation goal
  - If everything works, you have replaced the default navigate-to-goal behavior tree, so the turtlebot should ignore your goal and instead move in a square the number of times specified by your Repeat decorator